MIT/LCS/TR-221

ABSTRACT MODEL SPECIFICATIONS FOR DATA ABSTRACTIONS

Valdis Andris Bērziņš

This blank page was inserted to preserve pagination.

Abstract Model Specifications for Data Abstractions

by

Valdis Andris Berzins

July 1979

This research was supported by the National Science Foundation
under grants DCR74-21892 and MCS74-21892 A01.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts
02139

Abstract Model Specifications for Data Abstractions

by

Valdis Andris Bērziņš

Submitted to the Department of Electrical Engineering and Computer Science
on July 5, 1979 in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy.

Abstract

A data abstraction introduces a data type with a hidden representation. Specifications of data abstractions are required to allow the data to be described and used without reference to the underlying representation. There are two main approaches to specifying data abstractions, the abstract model approach and the axiomatic approach.

This thesis is concerned with the problems of formalizing and extending the abstract model approach. A formally defined language for writing abstract model specifications is presented. The correctness of an implementation with respect to an abstract model specification is defined, and a method for proving the correctness of implementations is proposed.

Our formulation treats data abstractions with operations that can dynamically create new data objects, modify the properties of existing data objects, and raise exception conditions when presented with unusual input values.

Thesis Supervisor: Barbara H. Liskov
Title: Associate Professor of Computer Science and Engineering

Keywords: specifications, abstract models, data abstractions, data types, errors, exceptions, side effects, modification of shared data, verification

Acknowledgements

Special thanks are due to my thesis supervisor Barbara Liskov, whose encouragement, limitless patience, and sound advice kept me going. My readers Carl Hewitt and Ronald Rivest have offered many valuable suggestions for improving the structure and presentation of this work. Deepak Kapur and Bob Scheifler have participated in many arguments and discussions which have clarified the ideas presented here. Eliot Moss did a very careful job in proofreading a draft of this work. The excellent text processing facilities provided by the MIT Lab for Computer Science have been invaluable in producing this document. This research was supported by the National Science Foundation under grants DCR74-21892 and MCS74-21892 A01.

CONTENTS

1. Introduction

Specifications play an important part in the programming process, especially in the design and construction of large programs. It is generally accepted that large programs should be designed as systems of loosely coupled independent modules, so that each module can be designed and understood without reference to the other modules. A precise specification of the behavior of a module decouples the programs that use a module from the programs that implement the module, since programs that use the module depend on the specification of the module rather than on the implementation. The hope is that the specifications of a module will usually be simpler and more stable than the implementation of the module, so that the use of the specifications will make it easier to design, implement, and maintain the modules that make up a program. Specifications are also needed for program verification.

The research reported here is primarily concerned with specifications for data abstractions. A data abstraction consists of a set of data objects and a set of primitive operations on those objects. The objects of a data abstraction are treated as abstract indivisible entities, which do not have any directly accessible substructure. The objects of a data abstraction can be manipulated only by means of the primitive operations provided by the data abstraction.[1] The behavior of a data abstraction is completely characterized by the behavior of its primitive operations, and the observable properties of the abstraction are precisely those computable in terms of the primitive operations. Since the behavior of a data abstraction is

---

1. The only exception to this rule is the boolean abstraction. The host programming language may provide statements, such as the conditional, which make the flow of control depend directly on a boolean value. These statements are not primitive operations of the boolean abstraction, and they cannot be defined using only the primitive operations.

independent of the way in which the associated data objects are represented in any particular implementation, introducing data abstractions is one way of decomposing a program into independent modules [39, 40, 19, 4, 29]. The concept of representation independence is made more precise by the definition of behavioral equivalence of data models developed in Chapter 3, and it is the basis for the usual data type induction rule [53].

To specify the behavior of a data abstraction, it is sufficient to specify the behavior of each operation, since the only way to interact with the objects of a data abstraction is by means of the primitive operations. The problem of specifying the operations of a data abstraction differs from the problem of specifying procedures because the specification of a data abstraction must be independent of the way the associated data objects are represented in any particular implementation of the abstraction. There are two main approaches to specifying data abstractions, the abstract model approach and the axiomatic approach.

In the abstract model approach, an abstract representation for the data objects is defined, and the operations are specified in terms of the abstract representation. The representation is abstract because it is constructed from mathematically defined domains, rather than the built in data types of some programming language. The abstract representation should be chosen so that the operations can be defined as simply as possible. The representation used in the implementation of a data abstraction must often be chosen to optimize space or time efficiency, and may be quite different from the abstract representation. To prove the correctness of an implementation with respect to an abstract model specification, it is necessary to define the correspondence between the representation used in the implementation and the abstract representation.

In the axiomatic approach, the set of data objects is defined implicitly, by giving a set

of axioms relating the primitive operations. The axioms specify the relationships that must hold between the operations of a data abstraction, and any structure satisfying the axioms is taken to be an acceptable model for the abstraction. If the axioms are consistent, then there will be at least one structure satisfying the axioms. It is possible for many different structures to satisfy the same set of axioms. To establish the correctness of an implementation with respect to an axiomatic specification, it is necessary to show that the operations of the implementation satisfy the axioms. An excellent treatment of correctness proofs based on axiomatic specifications can be found in [37].

The work reported here is concerned mainly with formalizing and extending the abstract model specification technique. We present a formally defined language for writing abstract model specifications. A criterion for judging the correctness of an implementation of a data abstraction with respect to an abstract model specification is developed, along with a method for proving that particular implementations are correct with respect to specifications in our specification language. Both the specification language and the proof technique apply to situations where mutable data can be shared. Previous work on specifications has largely avoided the issues associated with shared mutable data. Our formulation provides an integrated treatment of data abstractions with operations that can dynamically create new data objects, modify existing data objects, or raise exception conditions when presented with unusual input values.

1.1 Previous Work

Most high level programming languages have a set of built in data abstractions. Languages that support user defined data abstractions have been developed, including SIMULA 67 [5], CLU [29], and ALPHARD [54]. In these languages a program using a data abstraction does not have to mention the representation of that abstraction, so that the implementation structure can be changed without affecting any of the applications programs using the abstraction.

Surveys of specification techniques for data abstractions can be found in [31] and in [28].

The abstract model approach is direct in the sense that the set of data objects associated with a data abstraction is explicitly constructed. References to early work on abstract model specifications can be found in [31]. The problem of proving the correctness of an implementation of a data abstraction with respect to an abstract model specification has been treated by Hoare in [18], and the problem of proving the correctness of programs using the objects and operations of a data abstraction has been treated by Shaw in [47]. In both cases, the specification language has been introduced informally, and shared data has been excluded. The problem of specifying the behavior of data shared by concurrent processes has been treated in the context of the actor formalism by Yonezawa in [55].

The abstract model approach is related to the denotational definition method for programming languages developed at Oxford by Scott and Strachey [49, 46], in which a mathematical model is defined for each of the constructs of a programming language, including the data domains. The major emphasis of the work at Oxford has been directed at issues other

than data abstractions. A formal treatment of a language with the potential for sharing mutable data can be found in [50], although the model makes little attempt to abstract from the storage representation of the data. A denotational definition of CLU, a language with facilities for constructing user defined data abstractions, can be found in [45].

The axiomatic approach is indirect in the sense that the set of data objects associated with a data abstraction is defined implicitly. There are several different axiomatic specification techniques, which are distinguished by the kind of logic in which the axioms are embedded.

Axiomatizations of data abstractions in first order logic can be found in [19]. A first order logical approach has also been used in the iota system [37] for constructing and verifying programs that use data abstractions.

A restricted form of axiomatic specification using only conditionals and equations has come to be known as the algebraic approach [56, 10, 7, 13, 9]. The name stems from a uniform method for constructing a canonical model for axiomatizations expressed in this form. The canonical model is a many sorted algebra which is unique up to isomorphism, and which is called the initial algebra. A system for verifying programs using data abstractions specified by algebraic specifications has been developed at ISI [35, 11, 36].

The problem of proving properties of programs that manipulate potentially shared mutable list structures has been treated by Burstall in [2]. Burstall follows a hybrid approach, by explicitly introducing a model and defining its behavior axiomatically. Proofs about programs that manipulate pointers have been treated by Suzuki and Luckham [51, 32].

An approach to defining programming languages combining aspects of the direct and indirect approaches is being developed by Schaffert [44]. Schaffert treats shared mutable data abstractly, and considers the problem of proving properties of programs using mutable data

abstractions.

1.2 Motiviations for this Work

The original aim of this research was to develop tools and techniques for increasing the level of confidence that a formal specification for a data abstraction does indeed capture the behavior intended by the designer. We started with the algebraic specification technique, as described by the work of Zilles [56] and Guttag [10].

After some preliminary investigation, it became clear that there were a number of phenomena associated with the data types actually used by programmers that could not be adequately described by this specification technique as it stood, notably the dynamic creation of data objects, changes of state of potentially shared data, and exception handling.

It also appeared to be difficult to produce a well formed algebraic specification for a new data abstraction, especially if the exact behavior required was not yet completely designed. In our experience, a typical attempt to design a data abstraction using axiomatic specifications runs as follows. After analyzing the problem, the operations of the data abstraction are determined, and the inputs and outputs of each operation are identified. When the intended behavior of each operation on a typical set of input values is fairly well understood, a preliminary axiomatization is constructed. The process of producing the preliminary axiomatization helps to pinpoint special cases and boundary values for the input domain, and the problem is analyzed further to determine appropriate behavior for the operation on unusual or ill formed input values. The axioms are examined in light of the new design decisions and are adjusted to conform. After a few iterations each of the axioms looks plausible when considered in isolation. At this point the axiomatization is examined for consistency and

completeness, often at the cost of considerable effort. Fairly often we have found such an axiomatization to be inconsistent, and less often to be incomplete. It was disturbing to find that plausible axiomatizations could be ill formed, and that the effort of producing a precise description of a seeming simple design decision could be quite large.

We also designed some data abstractions using abstract model specifications, and found that the process was much easier. One point was that inconsistencies in the design would usually surface immediately, because it would not be possible to define some operation so that it satisfied all of the informal constraints, while the usual result of trying to axiomatize an inconsistent set of design decisions was a inconsistent axiomatization, which was often difficult to recognize as inconsistent. Another point was that minor perturbations in the behavior of an operation were easier to describe for an abstract model specification than for an axiomatic specification. As long as the meaning of the abstract representation is not changed, a modification in the definition of one operation cannot affect the other operations, since in an abstract model each operation is defined in terms of its effect on the abstract representation. In an axiomatic specification the meanings of the operations are defined in terms of the relations between them, so that a change in an axiom can affect many operations.

While the above is a very subjective judgment, based on our personal experience with a fairly small set of examples, we found that other people trying to use axiomatic specifications in the design process had similar complaints. This motivated a more extensive investigation of abstract model specifications.

We found that previous work on abstract model specifications was largely informal, and that abstract model specifications were used without saying what the specification language was or what the specifications meant. Since abstract model specifications appeared to have

advantages from the point of view of design, establishing a precise mathematical formulation of the specification technique arose as a natural subgoal. In the process of pursuing this goal, it became apparent that dynamic creation of data objects, state changes, and exception conditions could be readily incorporated into the framework. At that time, existing work on axiomatic specifications did not address these issues, which kept cropping up in the design of programs. As a result, the direction of this research shifted to developing and extending the abstract model specification technique, and the original problem was set aside as a subject for future investigation.

1.3 Assumptions and Restrictions

In the interests of defining a problem that can be treated in depth in a reasonable amount of time, we have made some restrictions on the scope of our investigation. These restrictions are explicitly stated below. A more detailed discussion of the restrictions and the reasons for introducing them can be found in Appendix I.

We have not considered cases where mutable data is shared by concurrent processes, so that a model of a computation as a linear sequence of operations is sufficient for our purposes. We have assumed that each operation is deterministic, so that every computation produces a unique result. These assumptions lead to a simpler characterization of the observable behavior of a data abstraction than would otherwise be possible.

We have adopted a model of exception handling in which operations are terminated if they raise an exception condition. This restriction allows the behavior of an operation to be described independently of the behavior of exception handlers and exception handling mechanisms, and leads to a clean model of data behavior.

We have assumed that each operation depends only on the properties of the data objects passed to the operation as arguments. Operations that depend on global data or on own data (i.e., operations with internal state) are excluded by this assumption. Without such a restriction on the operations, systems must be treated where the behavior of a data object may be affected without applying any of the primitive operations associated with the data abstraction, and the concept of behavioral equivalence (see Chapter 3) must be reformulated. Since the behavior of such structures is not completely characterized by the behavior of the primitive operations, we do not accept them as well formed data abstractions.

1.4 Results of this Work

We have investigated the structure of mathematical models of data abstractions, developed a general framework for proving the correctness of implementations, and proposed a prototype specification language based on these results.

1.4.1 Mathematical Models

A specification can be viewed as a method for singling out the structures (or models) that exhibit the desired behavior from those that do not, and the meaning of a specification can be identified with the class of models consistent with the specification. This gives us a basis for judging whether or not two specifications in two different formalisms have the same meaning. The set of structures consistent with a given axiomatic specification contains precisely those structures in which all of the axioms are true. An abstract model specification defines a particular model exhibiting the desired behavior, and the class of all models consistent with the specification contains just those structures with the same externally observable behavior as the

standard model. In this work, we have formally characterized the aspects of the behavior of a data abstraction that are detectable by an external observer.

We have described two classes of algebraic structures, exception algebras and state machines, which can be used as models for data abstractions with exception conditions and with state changes. This work will be of interest to people wishing to extend the axiomatic technique to include exceptions and state changes, since it explores the kinds of structures that will have to be defined axiomatically.

1.4.2 Proof Techniques

In treating a range of behavior including object creation, mutation of data, and exceptions, we have found it necessary to reformulate the criteria for the correctness of a proposed implementation of a data abstraction, and to develop new techniques for proving the correctness of an implementation with respect to the new criteria. These techniques are of interest also to people who wish to verify mutable implementations of data abstractions with respect to axiomatic specifications.

1.4.3 Specification Language

We have developed a specification language for defining data abstractions based on abstract models. This language has been given a mathematical definition that is sufficiently formal to support mathematical proofs of properties of the specification, and of the correctness of implementations. We have made an effort to incorporate all of the features necessary for a practical specification language, rather than to define a language designed to facilitate proofs of meta-theorems about the specification language. We have intended this language to serve as a

prototype, which can be used as a guide for people designing practical specification languages. The language presented here has been designed primarily to be read and written by humans, rather than to be mechanically processed (e.g., by a program verification system). In some applications it may be desirable to use a more restricted language, in order to facilitate automatic theorem proving at the expense of making the specifications harder to construct.

1.5 Overview of Remaining Chapters

In Chapter 2 we explain the novel aspects of data behavior associated with exception conditions, dynamic allocation, and mutation of potentially shared data, and describe algebraic structures suitable for modeling that behavior.

In Chapter 3 we formally define the externally observable behavior of a data abstraction. The meaning of a data abstraction is associated with the class of all structures exhibiting the same externally observable behavior. The concept of a reduced model for a data abstraction is developed and explored.

In Chapter 4 we present a specification language for constructing models, along with a mathematical definition for the semantics of the language. Each well formed expression of the language denotes an algebraic model. The construction of the model is explained, and the requirements an expression must satisfy in order to be a well formed specification are established.

In Chapter 5 we state our basic definition of the correctness of an implementation, and develop a methodology for proving the correctness of an implementation with respect to a standard model for the data abstraction to be implemented. The methodology is illustrated by examples of correctness proofs. The basic definition depends on the material in Chapter 3,

while the examples use the language developed in Chapter 4.

Chapter 6 contains our conclusions, a comparison of the abstract model specification technique to the algebraic technique, and indications of directions for future research.

2. Modeling Data Behavior

We will define the behavior of a data abstraction by constructing a *standard model* exhibiting that behavior. A model is a mathematical structure containing *interpretations* for the objects and operations of the data abstraction. The externally observable behavior of a data abstraction consists of the results of all finite computations composed from the primitive operations of the data abstraction and yielding objects of other types.[1] An abstract model is used to specify the externally observable behavior of a data abstraction. All properties of a model that are not externally observable are irrelevant, in the sense that they do not influence whether or not a proposed implementation of a data abstraction is correct with respect to the standard model. We will say that two models are *behaviorally equivalent* if and only if they have the same externally observable behavior. If two structures are behaviorally equivalent then they are models of the same data abstraction. Behavioral equivalence is treated in depth in Chapter 3.

The standard model is intended to be a isomorphic image of the data abstraction as conceived by the designer: every object of the data abstraction imagined by the designer should correspond to a unique object in the standard model, and the correspondence should preserve the operations. The standard model of an abstraction can be identified with the structure conceived by the designer, thus bridging the gap between the inaccessible pattern in the

---

1. Except for the boolean abstraction, the only way to interact with the objects of a data abstraction is by means of the primitive operations, so that the only way to export any information from an abstract type is by means of the primitive operations yielding results of some other type. The interested reader may wish to compare this idea with the treatment of sufficient completeness in [10].

designer's head and a publicly accessible mathematical structure. A well designed standard model should be *reduced*: it should not be possible to delete an object from the model or to coalesce two distinct objects without affecting the externally observable behavior of the model. The concept of a reduced model is discussed further in Chapter 3.

In this chapter we will consider various aspects of the behavior of a data abstraction, and show how they can be modeled using algebraic structures, but first we have to briefly examine the internal structure of a data abstraction and the ways in which a set of data abstractions can be related to each other.

2.1 Types and Subordinate Abstractions

We will call a set of data objects subject to the same operations a *type*. The definition of a new data abstraction introduces a new type, the *principal type* of the abstraction. Each operation of the abstraction involves objects of the principal type, and often also objects of other types, which we will call the *subordinate types* of the data abstraction. For example, the set of integers is the principal type of the integer data abstraction, and the set of booleans is a subordinate type, because the integer abstraction has the operations = and < which map pairs of integers into booleans. Every type is the principal type of some unique data abstraction, known as the *defining abstraction* of the type. The primitive operations on the objects of a type are just the operations of its defining abstraction.

A model for a data abstraction must contain interpretations for the principal type and operations, and also for the subordinate types, because the operations involve objects of the subordinate types as well as of the principal type. Each of the subordinate types of a data abstraction d is the principal type of its defining abstraction d'. Thus we are usually dealing

with a set of related data abstractions, and with a set of related models for those abstractions. We will assume that systems of data abstractions are defined incrementally, where the definition of a model for a new data abstraction explicitly introduces an interpretation for its principal type, and where the interpretations for the subordinate types are taken from the models for the defining abstractions for those types. This construction guarantees that a type is not given two different interpretations in a single system of models. However, a bit of caution is required, because it may not always be possible to define the data abstractions in a system in an order such that the defining abstractions of the subordinate types of each data abstraction are defined before the data abstraction itself. For example, suppose that the fixed point number abstraction has an operation for converting fixed point numbers to floating point, and that the floating point abstraction has an operation for converting floating point numbers to fixed point, say by rounding. In such a case, floating point numbers are a subordinate type for the fixed point number abstraction, and vice versa, so that it is not possible to define both types in an order such that their subordinate types have been previously defined.

In order to make the idea of a hierarchically ordered set of type definitions more precise, we define the **direct subordinate** and **subordinate** relations as follows.

Definition 1 Direct subordinate relation.
If d_1 and d_2 are data abstractions, then d_1 is a **direct subordinate** of d_2 if and only if the principal type of d_1 is a subordinate type of d_2.

Definition 2 Subordinate relation
The **subordinate** relation is the transitive closure of the **direct subordinate** relation.

We would like the **subordinate** relation to be a well founded partial order, but this need not always be the case, because two data abstractions can be **subordinate** to each other, as in the

above example. However, if we group together all of the data abstractions that are mutually **subordinate** (i.e., take the quotient with respect to the largest equivalence relation contained in **subordinate**), then the **subordinate** relation does in fact induce a partial order on the groups (equivalence classes).

We will treat each group of mutually **subordinate** data abstractions as a single module. A model for such a module will have several principal types, one for each data abstraction. Modules correspond to the equivalence classes introduced in the previous paragraph. The **subordinate** relation for modules is always a partial ordering. This ordering is also well founded, because the set of data abstractions in any real system is finite. Since the ordering is well founded, we can use structural induction with respect to the **subordinate** relation on modules when proving properties of systems of data abstractions (i.e., to establish a property for the data abstractions in the module *m*, we can assume the property holds for all abstractions **subordinate** to *m*).

It will usually be the case that each module defines a single data abstraction, with a single principal type. In the following discussion we will often tacitly assume that each model has only a single principal type, although the formal definitions will be formulated to deal with any number of principal types per model.

2.2 Simple Abstractions

The purpose of a standard model specification is to provide an interpretation for each type and for each operation of the data abstraction it specifies. A well chosen standard model should provide interpretations that are clean and simple. The most suitable modeling structures depend on the kinds of behavior that must be described. The simplest case is a data

abstraction without any exception conditions or any time dependent behavior, because in such a case the types can be interpreted as fixed sets of constant values, and the operations can be interpreted as functions on those sets. We will refer to this kind of abstraction as a *simple* data abstraction. The early work on algebraic specifications for data abstractions [56, 10, 7] dealt primarily with simple abstractions. Following their lead, we will model simple abstractions as heterogeneous algebras [1].

A heterogeneous algebra, also known as a many sorted algebra, is a pair $\langle P, F \rangle$, where $P = \{ P_\alpha \mid \alpha \in A \}$ is an indexed set of phyla (also called carriers), and where $F = \{ F_\beta \mid \beta \in B \}$ is an indexed set of operations. The index sets A and B contain the names of the types and operations, respectively. Each phylum in P is a set of data objects. Each operation in F is a function $F_\beta : P_{a(\beta, 1)} \times \ldots \times P_{a(\beta, n(\beta))} \rightarrow P_{r(\beta)}$, where $n : B \rightarrow \mathbf{N}$, $a : B \times \mathbf{N} \rightarrow A$, and $r : B \rightarrow A$ are functions such that $n(\beta) \geq 0$ is the number of arguments for F_β, $a(\beta, k)$ is the type index for the k-th argument of F_β, and $r(\beta)$ is the type index for the return value of F_β, and where $\mathbf{N}$ is the set of all natural numbers. The principal and subordinate types of a constant data abstraction are interpreted as the phyla of the algebra, and the operations of the data abstraction are interpreted as the operations of the algebra.

Simple data abstractions are easy to describe, but they represent a very restricted class of abstractions, which almost never occur in practice. For example, the fixed point number abstraction, a common and relatively simple data abstraction, fails to qualify as a constant data abstraction on two counts. First, an attempt to divide by zero results in an exception condition. Second, fixed point numbers have a *print* operation, which modifies the state of an output stream. Exception conditions and state changes are discussed in detail below.

2.3 Exception Conditions

Many programming languages have data abstractions with operations that may signal errors or raise *exception conditions* (we prefer the latter term). A common example is the integer data abstraction, where an attempt to divide by zero results in an exception. In general, an operation should raise an exception whenever it is called with an argument outside its natural domain of definition. Situations like this are quite common, so that it is important to include exceptions in our model of data abstractions.

2.3.1 Termination vs. Resumption

An exception causes a departure from the normal flow of control, to execute a program fragment intended to *handle* the exceptional condition. In cases where the exception handler can recover from the exception, the computation may continue, and otherwise it must be aborted. There is no universally accepted model for this process.

One viewpoint, which we shall adopt, is that an operation may have a number of return points, one for the normal case, and one for each exception. We shall refer to this viewpoint as the termination model of exception handling. According to the termination model, raising an exception is just a special way of terminating an operation.

An alternative viewpoint, which is commonly held, is that an exception causes the exception handler to be invoked as a procedure, with the implication that the operation that raised the exception will continue after the handler returns. We will refer to this viewpoint as the resumption model of exception handling.

Both alternatives have been implemented. For example, in CLU an exception

conditions always terminates the operation that raised it, while in PL/I the operation is resumed (for one class of exception conditions). A detailed analysis and comparison of the termination and resumption models can be found in [30], where it is argued that the termination model has a much simpler behavior than the resumption model.

2.3.2 Termination Conditions

We will assume that an operation of a data abstraction may terminate in any of a number of *termination conditions* [cf. 43], one of which (the **normal** condition) corresponds to the normal behavior of the operation, while the others correspond to the exception conditions that may be raised by the operation. The effects of an operation and the number and types of return values will usually depend on the termination condition. For motivational purposes, we will assume that when an exception occurs, the data objects produced by the operation, if any, are passed to the appropriate exception handler as arguments.[2]

A specification for a data abstraction with exceptions must therefore specify when each exception occurs, and what the results of the operation are for each termination condition. The definition of the host language must specify which error handler is associated with each occurrence of an exception, and what happens after the handler terminates. The only constraint we impose on the host language is that whenever an operation raises an exception, the operation is terminated before the handler is invoked, and may not be restarted.[3]

---

2. This corresponds closely to the exception mechanism in CLU. In other languages, more roundabout methods may have to be used for passing information to an exception handler, such as assigning values to global variables.
3. This constraint is implicit in [10] and [8].

2.3.3 Exception Algebras

In order to get a class of structures suitable for modeling data abstractions with exceptions, we have to extend the notion of a heterogeneous algebra. In a heterogeneous algebra as described in [1], each operation is a function whose range is some phylum of the algebra, but a typical operation of a data abstraction may return more than one data object, and it may return objects of different types in different termination conditions. Rather than introducing phyla with a complicated substructure, we prefer to relax the constraint on the allowable ranges of the operations, since we would like to maintain a simple correspondence between the types of a data abstraction and the phyla of the modeling structure. In an exception algebra, the range of a typical operation is the disjoint union of a family of sets, each of which is a cartesian product of some number of phyla (possibly zero[4]).

We will also include the index sets and the functions describing the types of the operations as explicit components of the exception algebra, to prevent confusion in situations where we are dealing with several algebras in the same context.

Definition 3 Exception algebra

An exception algebra is a tuple $\langle$ phyla : P, operations : F, arglength : n, argtype : a, tc : t, rlength : m, rtype : r, typenames : A, opnames : B, tcnames : T, pt : D $\rangle$, where $P = \{ P_\alpha \mid \alpha \in A \}$ is an indexed set of phyla, and where $F = \{ F_\beta \mid \beta \in B \}$ is an indexed set of operations, such that each operation in F is a function $F_\beta : P_{a(\beta, 1)} \times \ldots \times P_{a(\beta, n(\beta))} \rightarrow \sqcup \{ R_\tau \mid \tau \in t(\beta) \}$, where $\sqcup$ denotes the disjoint union operation, and where $R_\tau = P_{r(\beta, \tau, 1)} \times \ldots \times P_{r(\beta, \tau, m(\beta, \tau))}$. $n : B \rightarrow \mathbf{N}$, $a : B \times \mathbf{N} \rightarrow A$, $t : B \rightarrow \mathcal{P}(T)$, $m : B \times T \rightarrow \mathbf{N}$, $r : B \times T \times \mathbf{N} \rightarrow A$ are functions such that $n(\beta)$ is the number of arguments for F_β, $a(\beta, k)$ is the type index for the k-th argument of F_β, $t(\beta)$ is the set of all termination conditions that may result from

---

4. The empty cartesian product is a singleton set containing the empty sequence.

F_β. $m(\beta, \tau)$ is the number of objects returned by F_β in the termination condition τ, and $r(\beta, \tau, k)$ is the type index for the k-th return value of F_β in the termination condition τ. A is the set of type names, B is the set of operation names, T is the set of termination condition names, and $D \subseteq A$ contains the names of the distinguished principal types. $\mathbf{N}$ is the set of natural numbers.

The details of this formal definition of an exception algebra will be used primarily in Chapter 3, and in the proofs of the theorems in Appendix III. The following example may help to clarify the meaning of the various components of an exception algebra. Let A be an exception algebra model for the integer data abstraction. Then we have:

A.typenames = { "int", "boolean" }
A.opnames = { "plus", "times", "difference", "quotient", ... }
A.tcnames = { "normal", "zero_divide" }
A.pt = { "int" }
A.phyla$_{\text{int}}$ = { 0, 1, -1, 2, -2, ... }
A.phyla$_{\text{boolean}}$ = { T, F }
A.operations$_{\text{plus}}$ = { $\langle x, y, z \rangle \mid z = x + y$ }
...

Quotes have been used to emphasize that the first four sets contains names (strings) rather than the sets they denote. Note that an algebra is a labeled tuple, and that we are using a dot notation similar to that used for the components of records in PASCAL to refer to the components of the tuple. If A.arglength = n, A.argtype = a, A.tc = t, A.rlength = m, and A.rtype = r, then:

n(*quotient*) = 2,
a(*quotient*, 1) = a(*quotient*, 2) = "int",
t(*quotient*) = { **normal**, zero_divide },
m(*quotient*, **normal**) = 1,
m(*quotient*, zero_divide) = 0, and
r(*quotient*, **normal**, 1) = "int".

In the specification language described in Chapter 4, we will describe the type information for an operation in a compact syntax illustrated below for the *quotient* operation.

quotient: int × int → ⟨ normal : int ⟩ + ⟨ zero_divide : ⟩

The range of an operation, which is a disjoint union, is written as the sum of the components for each termination condition. The component corresponding to the termination condition τ is written as $\langle \tau : R_\tau \rangle$, where the normal component may be abbreviated by dropping the angle brackets, the colon, and the condition name.

The reader should note that termination conditions and data objects are treated in different ways, and that the inputs to an operation are always ordinary data objects, which are never used to represent exceptions. In previous work on specifying data abstractions with exceptions, exceptions were modeled as distinguished *exception objects*, which were either elements of extra phyla [10] or distinguished subsets of the ordinary phyla [8]. We have followed [43] in introducing explicit named termination conditions, maintaining a distinction between termination conditions and data objects, since we feel that this approach provides a more coherent and disciplined view of the exceptions associated with a data abstraction.

2.4 Time Dependent Behavior

Many programming languages have data abstractions with data objects whose properties may be changed. Two common examples are records in PASCAL and arrays in extended LISP. Since data abstractions with time dependent properties are in fact widely used, it is important to develop a formalism suitable for specifying their behavior.

An operation is non-functional if it is possible to invoke the operation with the same

arguments at two different times and get two distinguishable results. A data abstraction exhibits time dependent behavior if it has at least one non-functional operation. Data abstractions with time dependent behavior will be modeled as *state machines*. A state machine is a special kind of exception algebra containing a distinguished phylum of *system state functions*. The progression of time in a computation is represented by the sequence of system states of the state machine.[5]

We distinguish two kinds of time dependent behavior. If an operation changes the properties of an existing data object, we will say that the operation *mutates* the data object. If a data abstraction has no operations that mutate any data objects, then the abstraction is *immutable*, and otherwise it is *mutable*. If every invocation of an operation returns a data object that is distinguishable from all data objects that have been computed previously, we say that the operation *creates* a new data object. If a data abstraction has no operations that create new data objects, then the abstraction is *static*, and otherwise it is *dynamic*. It is possible for a dynamic data abstraction to be immutable, as illustrated by the unique id abstraction described in Chapter 4.

Mutable data abstractions are usually dynamic, since the possibility of sharing data objects goes hand in hand with the need to create new data objects. A change in the state of a mutable data object is visible in all contexts in which the data object appears. If all of the contexts in which a given data object is used are not known, as is often the case in a program, then the data object cannot be mutated without risk of violating the assumptions made about

---

5. We are relying on our assumption that a computation is a single sequential process. The history of a parallel computation has been described as a partially ordered set of local states in [55].

the data object in some of the other contexts in which it may appear. A newly created data object is known to occur only in the context in which it was created, and can therefore be mutated without risk of interfering with other parts of the program.

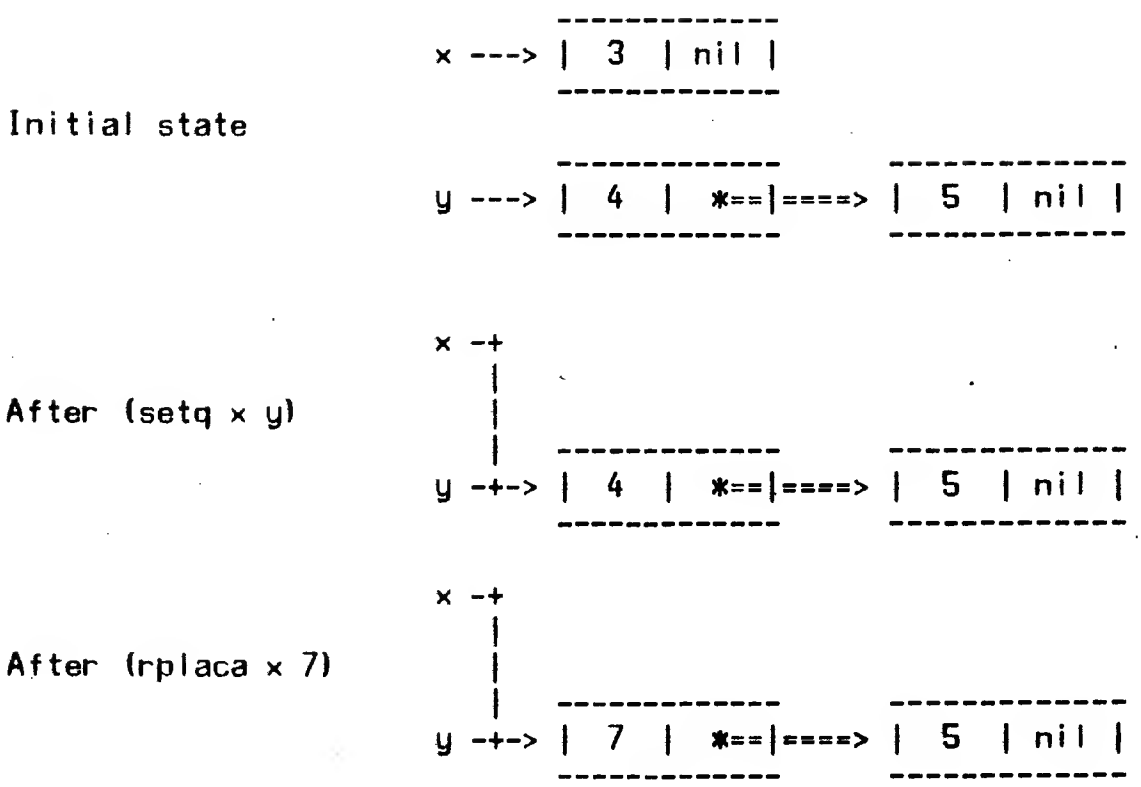
Data abstractions that are mutable or dynamic will be modeled as state machines, since they exhibit time dependent behavior. Data abstractions that are both static and immutable can be modeled as exception algebras without introducing states. The rest of this section is concerned with state machine models.

2.4.1 Data Objects vs. Variables

In the early work on abstract data types, abstract data objects were treated as immutable values, and all changes of state were identified with assignments of new abstract values to program variables. This point of view is now widely held, and is often taken for granted in work on specifications for data abstractions. However, as clearly stated in Hoare's pioneering paper [18], this approach is not suited for describing programs that manipulate pointers, or more abstractly, for describing mutable data abstractions that allow sharing of mutable data objects.

The distinction between the assignment of new values to variables and mutation of data becomes important in cases where mutable data is *shared* (several variables denote the same data object). Consider the example from LISP illustrated in Figure 1. Suppose that initially the value of the variable x is the list (3) and the value of the variable y is the list (4 5). The assignment (*setq* x y) will change the value of x to be the list (4 5) which is identically the same list as the initial value of y. This assignment has not influenced the properties of the lists (3) or (4 5), and therefore has not affected any other variables whose values happen to be the

Figure 1. Shared Mutable Lists



---

same lists as the original values of x or y. If we now modify the list x by executing the operation (*rplaca* x 7), we will have changed the first element of the list x to be 7. Both x and y continue to denote the same list (the original value of y), but the first element of this list has changed, so that the value of either x or y would print out as (7 5). Whenever a data object is modified, that change is visible in all variables that denote the data object, and in all other data objects that refer to (or "contain") the modified object.

The classical approach of associating all changes with the variables does not work very well in cases where mutable data is shared. If we were to insist that list values be modeled

as immutable sequences, and that all changes be described by assigning new values to the variables, then we would have a situation where a *rplaca* operation could change the values of arbitrarily many variables, depending on how the data was shared. By associating states with the data objects themselves rather than with the variables, we can overcome this difficulty, since changes can be localized in an object centered description. An example of a description of a mutable data structure with shared subcomponents can be found in Section 2.4.4.

The treatment of potentially shared mutable data has been one of the major goals of this work. Our approach is most closely matched to object oriented languages such as CLU and LISP, and our work is more or less applicable to languages with pointers and heap allocation, such as Euclid, Algol 68, and PL/I. We treat operations as functions that take a system state and some data objects, and produce a new system state and some data objects. The variables of the host programming language do not explicitly enter into our treatment, and we leave a discussion of the assignment of data objects to variables to the definition of the host programming language. Our treatment is directly applicable to the programming language CLU, in which the invocation of an operation or procedure may change the properties of some data objects, but is guaranteed not to disturb the association between variables and data objects. For host programming languages where the invocation of a procedure may alter the association between variables and data objects, as in (impure) LISP, Euclid, Algol 68, and PL/I, a correspondence has to be made between the operations of the language and the operations of the abstract model for each data abstraction.

There are two ways of incorporating abstractions with operations that assign to their input parameters in our framework. One way is to consider the abstractions to be immutable, with operations that return vectors of values to be assigned to the output variables of the

procedure. Another way to model such operations is to consider the L-values [cf. 50] of the variables to be part of the data object rather than the variable, and to treat the data abstraction as mutable.

The first approach is well suited in cases where there is no sharing of mutable data. Aliasing can in fact introduce sharing between the formal parameters of a call-by-reference procedure, so that special care is required in cases where the same variable is passed in more than one argument position [17].

In order to describe data objects whose properties are subject to change, we will introduce a system state function, which maps each data object into its properties in the current state. Only the permanent properties of a data object are represented by the interpretation of a data object in a state machine model, while the properties of a data object that are subject to change are represented by the image of the object under the system state function. For most mutable data abstractions, the only permanent property of data object is its identity.

2.4.2 State Machines

Mutable data abstractions are modeled as state machines, which are defined formally below. A state machine is an exception algebra with a distinguished phylum of system states.

Definition 4 State Machine

A state machine is a tuple $\langle$ phyla : P, operations : F, statefunctions : Σ, states : Δ, arglength : n, argtype : a, tc : t, rlength : m, rtype : r, typenames : A, opnames : B, tcnames : T, statenames : S, ss : s, pt : D $\rangle$, where $P = \{ P_\alpha \mid \alpha \in A \}$ is an indexed set of phyla, and where $F = \{ F_\beta \mid \beta \in B \}$ is an indexed set of operations, such that each operation in F is a function

$F_\beta : P_s \rightarrow (P_{a(\beta, 1)} \times \ldots \times P_{a(\beta, n(\beta))} \rightarrow P_s \times \sqcup \{ R_\tau \mid \tau \in t(\beta) \})$,

where $\sqcup$ denotes the disjoint union operation, and where $R_\tau = P_{r(\beta, \tau, 1)} \times \ldots \times P_{r(\beta, \tau, m(\beta, \tau))}$. $\Sigma = \{ \Sigma_\alpha \mid \alpha \in S \}$ and

$\Delta = \{ \Delta_\alpha \mid \alpha \in S \}$ are indexed sets such that each $\sigma_\alpha \in \Sigma_\alpha$ is a state function $\sigma_\alpha : P_\alpha \rightarrow \Delta_\alpha$. If $\sigma \in P_s$ then $\sigma = \cup \{ \sigma_\alpha \mid \alpha \in S \}$ for some $\sigma_\alpha \in \Sigma_\alpha$. $n : B \rightarrow \mathbf{N}$, $a : B \times \mathbf{N} \rightarrow A$, $t : B \rightarrow \mathcal{P}(T)$, $m : B \times T \rightarrow \mathbf{N}$, $r : B \times T \times \mathbf{N} \rightarrow A$ are functions such that $n(\beta)$ is the number of arguments for $F_\beta(\sigma)$ for any system state σ, $a(\beta, k)$ is the type index for the k-th argument of $F_\beta(\sigma)$, $t(\beta)$ is the set of all termination conditions that may result from $F_\beta(\sigma)$, $m(\beta, \tau)$ is the number of data objects returned by $F_\beta(\sigma)$ in the termination condition τ, and $r(\beta, \tau, k)$ is the type index for the k-th return value of $F_\beta(\sigma)$ in the termination condition τ. A is the set of type names, B is the set of operation names, T is the set of termination condition names, $D \subseteq A$ contains the names of the principal types, $S \subseteq A$ contains the names of the types that have a corresponding set of state functions, and $s \in (A - S - D)$ is the distinguished phylum of system states. $\mathbf{N}$ is the set of natural numbers.

The phylum of system states P_s contains all possible system state functions, one of which represents the current global state. A system state function is the disjoint union of all the individual state functions, each of which represents the current state of some type with time dependent behavior. The disjoint union of a family of functions $f = \cup \{ f_i \mid i \in \mathbf{I} \}$, where $f_i : d_i \rightarrow r_i$, is a function $f : \cup \{ d_i \mid i \in \mathbf{I} \} \rightarrow \cup \{ r_i \mid i \in \mathbf{I} \}$ such that whenever $x \in \cup \{ d_i \mid i \in \mathbf{I} \}$ and $x = \langle i, y \rangle$, $f(x) = f_i(y)$. Informally, the elements of the domain of the system state function are tagged with the name of the phylum they came from, so that the same set can be used to represent many different phyla without causing any interference among the various components of the system state function. The sets Σ_α and Δ_α are the domains and ranges of the individual state functions, and hence are used in the construction of the phylum of system states P_s, but they are not themselves phyla of the state machine, reflecting the fact that none of the operations of the state machine take individual state functions or individual data states as arguments. The set of statenames S specifies which phyla represent mutable types. Individual state functions are associated only with the mutable types of a data abstraction.

The operations of the state machine are curried,[6] so that formally an operation of a state machine is viewed as a family of operations parameterized by the current system state. This structure is introduced because the system state is qualitatively different from the other arguments of a typical operation, and because this structure makes corresponding notations for state machines and exception algebras more uniform. The operations of any immutable subordinate types are extended to take the system state as an extra argument, and to return the unchanged system state as an additional return value (the first component of the tuple of return values).

Each operation of a state machine takes the current system state as its first argument, and when supplied with the rest of its arguments the operation produces the new system state as its first return value. The reason for making the global state an argument to each operation of a data abstraction, rather than just the state function of the principal type, is that the operation may depend on or modify the state of some subordinate type. A common example of this kind of behavior is the *print* operation of a data abstraction, which modifies an output stream, but which usually does not affect the state of any data object belonging to the principal type.

If none of the principal types of a data abstraction has an associated phylum of state functions, then we will say that the abstraction *introduces no mutability*. An abstraction that introduces no mutability may still exhibit time dependent behavior, and hence require a state machine model, if it has some operations that depend on or modify the state of some subordinate type, or if it has some operations that take or return mutable data objects.

---

6. The process of abstracting from a function with n arguments to a higher order function which takes one argument and returns a function of n-1 arguments is named after Haskell B. Curry [3].

2.4.3 Mutation of Data Objects

In a state machine, the properties of a data object may depend on the current system state. Typically the objects of a mutable type are modeled as tokens without any attributes except for their identity. The purpose of the state function is to associate the current data state of each data object with that object, so that the same object can have different properties in different states. The set of *data states* Δ_α for the type α is the range of any individual state function $\sigma \in \Sigma_\alpha$ for that type. The data state of a mutable object is roughly analogous to the representation of an immutable object in an exception algebra. In an exception algebra model, the properties of a data object are computed in terms of some representation structure, while in a state machine model, the properties of a data object are computed in terms of the representation and its images under the state function.

A very simple example of a mutable data abstraction is the integer cell. An integer cell is a unit of memory that can store an integer value. A model for integer cells can be constructed by using the natural numbers for $P_{\text{integer cell}}$ and the integers for $\Delta_{\text{integer cell}}$ since the only observable property of a cell that is subject to change is the identity of the integer currently contained in the cell. The system state function σ maps every natural number representing a cell into the integer that is the current contents of that cell. There are three operations on integer cells: *create*, *fetch*, and *store*. The *create* operation creates a new cell with a specified integer as its initial contents. (The creation of data objects is discussed in Section 2.4.5.) The *fetch* operation applies the state function to the cell to get its current contents, and the *store* operation produces a new system state that differs from the old one only in mapping the updated cell into its new contents. A language for specifying models is defined in Chapter

4, and a number of complete examples of models for mutable data abstractions can be found there.

2.4.4 Sharing of Mutable Data

From the point of view of this work, the existence of sharing relationships among immutable data objects is not externally observable, since we are concerned only with the results of a computation, and not with the time and space requirements for performing the computation. A specification of an immutable data abstraction can therefore be constructed without considering potential sharing relationships. Sharing relationships among mutable data objects are often externally observable, so that they must be described in a state machine model, at least to the extent that they influence the externally observable behavior of the abstraction.

To reflect possible sharing relationships, the set of data states is allowed to overlap with the phyla of a state machine, so that the data state of an object x may be or may contain another object y that lies in the domain of the system state function, and therefore has a data state of its own. This kind of modeling structure is indicated whenever the object x has a potentially shared subcomponent y, such that the state of y is subject to change and such that the externally observable behavior of x depends on the state of y.

In the general case, the behavior of a data object x may depend on an indefinitely large set of data states, which are reachable from x by repeatedly applying the current system state to x and to components of other data states already in the set. We will call this set the *reachability closure* of the object x. For data abstractions where there are no externally observable sharing relations, as in the integer cell example, the set of data states should be chosen to be disjoint from the domain of the system state function, so that all of the state

information is reachable by means of a single application of the system state function.

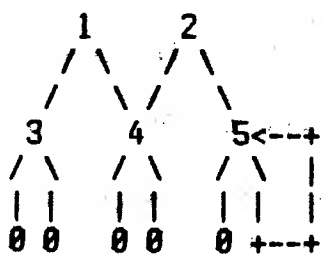
Mutable binary graphs are a classic example of a data abstraction where sharing relationships are important. This abstraction has operations for creating the null graph, for creating a composite graph with given left and right subgraphs, for extracting the left and right subgraphs of a composite graph, for modifying the left and right subgraphs of a composite graph, and predicates for testing if a graph is empty and if two graphs are identical. One way to construct a state machine model for binary graphs is to take $P_{\text{binary graph}}$ to be the set of natural numbers N, and $\Delta_{\text{binary graph}}$ to be the disjoint union null $\cup$ (N $\times$ N). The data state of a graph is either null, indicating that the graph is empty, or it is a pair of natural numbers representing the left and right subgraphs. An illustration of a system state σ containing a number of overlapping binary graphs is shown in Figure 2. Note that the graph represented by the number 4 is a subcomponent of the graphs 1 and 2, and is therefore shared. Binary graphs can also contain cycles, as shown by graph 5, which is its own left subgraph.

The mutation of shared data is a phenomenon that has been avoided in most existing work on specifications for data abstractions. As the example in Figure 2 indicates, it is not difficult to describe shared mutable data once we adopt a point of view centered on data objects

---

Figure 2. Shared Binary Graphs

$\sigma(0)$ = null
$\sigma(1)$ = ⟨3, 4⟩
$\sigma(2)$ = ⟨4, 5⟩
$\sigma(3)$ = ⟨0, 0⟩
$\sigma(4)$ = ⟨0, 0⟩
$\sigma(5)$ = ⟨0, 5⟩



rather than on variables. Some of the issues involved in reasoning about shared mutable data will be discussed in Chapter 5.

2.4.5 Creation of Data Objects

The principal type of a data abstraction is a fixed set for both static and dynamic data abstractions. For a dynamic data abstraction, the principal type is the set of all data objects of the given type that *can* be created by any finite computation in terms of the primitive operations of the data abstraction.

The *population* of a dynamic data abstraction d in a system state σ is the set of all objects of the principal type of d that exist in the state σ. The concept of a population is meaningless for static data abstractions. Since we find it convenient to work with total functions, we adopt the convention that the data state of any object that has not been created yet is the special object **undefined**, which is a member of every phylum of the state machine. All operations of the state machine are implicitly extended to apply to this extra object by the following strictness requirement:

$$\forall i \, [\, 1 \le i \le n \;\&\; x_i = \textbf{undefined} \Longrightarrow f(x_1, \ldots, x_n) = \textbf{undefined} \,]$$

for any operation f taking n arguments, for any $n \ge 1$. We also adopt the convention that $\sigma(\textbf{undefined}) = \textbf{undefined}$ for every system state $\sigma \in \Sigma$.

Definition 5 **Population of a data abstraction.**
The population of the data abstraction d in the system state σ is defined to be the set $\{\, x \in P_d \mid \sigma(x) \neq \textbf{undefined} \,\}$.

We will assume that in the initial system state every mutable type has an empty

population, and that objects are added to the population as they are created. We will also assume that every data object must be computed (i.e., returned as the value of some operation) before it can be used as an argument to a subsequent operation.

We would like any program we can write in terms of the primitive operations of a data abstraction to be guaranteed to return only data objects with a well defined state, and we will call a data abstraction *secure* if it has this property.

Data abstractions with operations that explicitly destroy data objects can be modeled readily in our framework, by having the operation change the state of the data object it is to destroy back to the original value **undefined**, thus removing it from the current population. Data abstractions with operations that explicitly destroy data objects cannot be secure, because a computation that creates an object, destroys it, and then applies any further operation to it will produce **undefined** as a value. The problem of deciding when it is safe to explicitly destroy a given data object must thus be addressed anew for each program that uses objects of an insecure data abstraction. This is known as the dangling reference problem, and it is generally acknowledged to be difficult.

We will concern ourselves mostly with secure data abstractions. The population of a secure data abstraction grows monotonically, and the reachability closure of any object in the population of a secure data abstraction will never contain the data object **undefined**.

Informally, we will say that a model is *reduced* if it does not contain any unnecessary data objects. (A more careful definition of a reduced model can be found in Section 3.3.) The standard model of a data abstraction should be reduced, since this generally leads to a cleaner specification. In the context of a state machine model, this means that an operation should extend the population only when it creates a "new" abstract object. An abstract object is "new"

if it is distinguishable from every object in the old population by means of some finite sequence of operations. In practice the required sequence of operations is often very easy to find, since many dynamic data abstractions provide an *equal* operation which can be used to test if two abstract objects are identical.

A very simple example of a dynamic data abstraction is the unique id abstraction, which has only two operations, *create* and *equal*. The *create* operation creates new unique ids; a newly created unique id is unique because it is guaranteed to be distinct from all previously created unique ids. The only way to create a unique id is by means of the *create* operation. The *equal* operation is provided as a means of comparing unique ids, and it is guaranteed to distinguish a newly created unique id from any previously existing unique id. Unique ids are immutable (so that they cannot be forged or tampered with - one application for unique ids is in implementing capability based data protection schemes).

This example illustrates that there is a state change associated with the creation of a new data object, as reflected by the increased size of the population, even though the properties of all previously existing objects may be unchanged. Note that the *create* operation is not a function of its arguments unless the state is explicitly included as an argument to the operation, because it will return different unique ids in different states, and it will never return the same one twice.

Another example of a dynamic data abstraction is the impure list abstraction (as found in LISP), with the operations *cons*, *car*, *cdr*, *atom*, *equal*, *eq*, *rplaca*, and *rplacd*. Each time it is called, the *cons* operation constructs a new list, which is distinguishable from any previously existing list by means of the *eq* operation. The impure list abstraction is also mutable, because the *rplaca* and *rplacd* operations can be used to modify the contents of existing lists. These

operations can also be used to distinguish a newly created list from a previously existing list with the same contents, by modifying one of the lists and looking to see if the other is changed also. If the lists are distinct, then one will be changed and the other will not be. Thus the impure list abstraction would be dynamic even without the *eq* operation. In the general case, two abstract objects are identical only if they have the same observable properties in the current state, and if they are guaranteed to have the same properties in all subsequent states.

Consider a restricted kind of list, which has the same operations as the impure lists of the previous example, except for *eq*, *rplaca*, and *rplacd*. This list abstraction is immutable, and also static, because there is no way to distinguish the list returned by one invocation of *cons* from that returned by a later invocation with the same arguments. This example demonstrates that whether or not a given operation returns a new abstract data object depends on the other operations of the abstraction. It may require a bit of thought to decide if a given data abstraction is in fact dynamic, and hence requires a state machine model, or if it is static and immutable, and hence should be specified by an exception algebra model.

3. Denotations for Data Abstractions

The meaning (or denotation) of a data abstraction is the class of all models of the data abstraction. In the axiomatic approach to specifying data abstractions, this class is taken to be the class of all models satisfying a given set of axioms. In the abstract model approach, the class of all models of a data abstraction is taken to be the set of all models with the same observable behavior as a given model, which is explicitly constructed.

In this work, we will assume that a model for a data abstraction is an exception algebra. We will say that a model is *dynamic* if it has a distinguished phylum of system states, and that it is *static* if it does not.

3.1 Complete and Partial Models

A model for a data abstraction d is *complete* if and only if d has interpretations for the types and operations of d and of every data abstraction **subordinate** to d. The externally observable behavior of d is characterized by the finite computations in terms of the operations of d and the abstractions **subordinate** to d, and any such computation can be interpreted in a complete model for d. A *partial* model for d may leave some of the abstractions **subordinate** to d uninterpreted.

Since the identities of the objects in a model are not *a priori* observable, there may be no way to compare the results of a closed computation in two different models. This problem is resolved by insisting on a unique standard model for the booleans, containing exactly two boolean values, so that the results of any computation producing a boolean value can be compared for any set of models. To reduce all comparisons of results to the problem of

comparing boolean values, it is necessary to include the operations of the **subordinate** abstractions in the computations. Thus complete models are required to make sure that every computation of interest can be interpreted.

In practice, a system of data abstractions is described incrementally, by giving a partial description for each abstraction (or set of mutually **subordinate** abstractions) d in the system. The partial descriptions give a prescription for constructing interpretations for the principal type and operations of d, assuming that complete models for the abstractions **subordinate** to d are already defined. In particular, the interpretations of the subordinate types of d are to be taken from the models for the defining abstractions of those types. The construction of a complete model for d is described more precisely below.[1]

Let d be a data abstraction, let $d_1, \ldots, d_n$ be the abstractions **subordinate** to d, and let m_i be a complete model for d_i for each i in the range $1 \le i \le n$. Suppose we have a partial description D for d, which gives the signature of d, the name of the principal type of d, and interpretations for the principal type and operations of d. If D describes an exception algebra, then a complete model m for d is constructed as follows.

$m.\text{phyla} = D.\text{phyla}_{D.\text{pt}} \sqcup \left(\bigsqcup_{1 \le i \le n} m_i.\text{phyla}_{m_i.\text{pt}} \right)$

$m.\text{ops} = D.\text{ops} \sqcup \left(\bigsqcup_{1 \le i \le n} m_i.\text{ops} \right)$

$m.\text{arglength} = D.\text{arglength} \sqcup \left(\bigsqcup_{1 \le i \le n} m_i.\text{arglength} \right)$

$m.$argtype, $m.$tc, $m.$rlength, and $m.$rtype are similarly defined as disjoint unions.

$m.\text{typenames} = D.\text{typenames} \cup \left(\bigcup_{1 \le i \le n} m_i.\text{typenames} \right)$

$m.\text{opnames} = D.\text{opnames} \sqcup \left(\bigsqcup_{1 \le i \le n} m_i.\text{opnames} \right)$

$m.\text{tcnames} = D.\text{tcnames} \cup \left(\bigcup_{1 \le i \le n} m_i.\text{tcnames} \right)$

---

1. The details of this construction are not essential for an understanding of the rest of this work, and may be skipped on a first reading.

$m.\text{pt} = D.\text{pt}$

where $\sqcup$ denotes disjoint union and where $\cup$ denotes ordinary set theoretic union. If D describes a state machine, then the above relations still apply, and we have to add the following:

$m.\text{statefunctions} = \{ D.\text{statefunctions}_{D.\text{pt}} \} \cup \{ m_i.\text{statefunctions}_{m_i.\text{pt}} \mid m_i \text{ is a state machine} \}$
$m.\text{states} = \{ D.\text{states}_{D.\text{pt}} \} \cup \{ m_i.\text{states}_{m_i.\text{pt}} \mid m_i \text{ is a state machine} \}$
$m.\text{statenames} = D.\text{statenames} \cup (\cup \{ m_i.\text{statenames} \mid m_i \text{ is a state machine} \})$
$m.\text{ss} = D.\text{ss}$
$m.\text{phyla}_{m.\text{ss}} = \{ \sqcup \{ \sigma_\alpha \mid \alpha \in S \} \mid \sigma_\alpha \in \Sigma_\alpha \text{ for each } \alpha \in S \}$
where $S = m.\text{statenames}$ and $\Sigma = m.\text{statefunctions}$.

In the rest of this Chapter, we will limit our discussion to complete models, and we will frequently leave out the qualifier "complete".

3.2 Behavioral Equivalence

Informally, two models are behaviorally equivalent if they have the same externally observable behavior. In this section we develop a precise mathematical definition of an equivalence relation that captures this informal notion. We define closed computations, and the interpretation of a closed computation in a model. Two models are behaviorally equivalent if they contain interpretations for the same types and operations, and if the value of any finite closed computation in one model is indistinguishable from the value of that computation in the other model.

Behavioral equivalence is an important notion, because it is the basis for defining the correctness of an implementation of a data abstraction. An implementation defines a model for the abstraction it implements, and the implementation is correct if the model it defines is

behaviorally equivalent to the model that specifies the abstraction.

We can meaningfully compare two models only if they have interpretations for the same types and operations. Two models can be behaviorally equivalent only if they have the same signature.

Definition 6 Signature

The signature of an exception algebra a is the tuple
⟨ arglength : a.arglength, argtype : a.argtype,
tc : a.tc, rlength : a.rlength, rtype : a.rtype,
typenames : a.typenames, opnames : a.opnames, tcnames : a.tcnames ⟩.

If two exception algebras have the same signature, then they have the same names for the phyla, operations, and termination conditions, and corresponding operations have the same numbers and types of arguments, the same set of possible termination conditions, and the same numbers and types of return values in each termination condition. As a matter of notational convenience, we require comparable models to be indexed by the same sets, so that corresponding types and operations have the same names, and we can talk about the interpretations of the same operation name in several different models.

In order to characterize the kinds of behavior a data abstraction may exhibit, we define the set of closed computations.

Definition 7 Closed computation

A closed computation with respect to a signature S is a finite sequence of pairs C such that
$C[i]$ = ⟨ op : f, args : s ⟩ for each i in the range $1 \le i \le$ length(C),
where $f \in S$.opnames, and s is a sequence of argument specifications such that
length(s) = S.arglength(f),
$s[j]$ = ⟨ step : n, tc : τ, result : k ⟩, (the source of the j-th argument to f)
$1 \le n < i$, (n is the index of a previous step of the computation)
$\tau \in S$.tc($C[n]$.op), (τ is the required termination condition for step n)

$1 \le k \le S.\,\mathrm{rlength}(C[n].\,\mathrm{op}, \tau)$, (the k-th object returned by step n must exist)
and $S.\,\mathrm{rtype}(C[n].\,\mathrm{op}, \tau, k) = S.\,\mathrm{argtype}(f, j)$ (and it must have the right type)
for each j in the range $1 \le j \le \mathrm{length}(s)$.

A closed computation is a sequence of steps, where each step is the application of some operation of a data abstraction to data objects resulting from previous steps. Every computation starts from nothing, and computes data objects as it proceeds. A closed computation is analogous to an uninterpreted flowchart, since the sequence of the operations is given, but the operation names are left uninterpreted. A step is a pair consisting of an operation name and a sequence of argument specifications. An argument specification is a triple, which specifies a previous step, a required termination condition for that step, and the index of the desired result. The index is necessary because an operation will in general return more than one object, and we have to say which of the returned objects to use. Since the number and types of objects resulting from an operation can be different for different termination conditions, an argument specification requires the step producing the argument object to terminate in a particular termination condition, so that we can be sure that the specified data object is of the proper type. A closed computation can fail to have an interpretation in a given model, if the termination conditions actually computed do not match the required termination conditions in the argument specifications of the closed computation.

An example of a closed computation $C1$ over the list abstraction of pure LISP is shown below.

$C1[1]$ = ⟨ op : *nil* , args : ⟨ ⟩ ⟩
$C1[2]$ = ⟨ op : *cons* , args : ⟨ ⟨ step : 1, tc : **normal**, result : 1 ⟩, ⟨ step : 1, tc : **normal**, result : 1 ⟩ ⟩ ⟩
$C1[3]$ = ⟨ op : *cons* , args : ⟨ ⟨ step : 1, tc : **normal**, result : 1 ⟩, ⟨ step : 2, tc : **normal**, result : 1 ⟩ ⟩ ⟩

This computation computes the value of the LISP expression "(*cons nil* (*cons nil nil*))".

A closed computation consists of a finite sequence of operations, with no conditionals or other control structures, and can be thought of as a trace of the execution of some program that uses the operations of the data abstractions of interest. The finite prefixes of the history of any program can clearly be described by a set of closed computations, and any finite closed computation can be described by a program in just about any programming language. Note that a machine for executing closed computations requires an unbounded amount of memory, because it is assumed that the results of each step are saved, and may be used in any number of succeeding steps.

We want to know whether or not there is some computation that yields observably different results when interpreted in each of the two models whose behavior we are comparing. It is sufficient for this purpose to consider only the finite computations: given two infinite sequences, if we know that their prefixes of length n are the same for every natural number n, then the original infinite sequences must be the same as well.

The interpretation of a computation in a given model is the sequence of results obtained by applying the interpretations of the specified sequence of operations in the model to the specified arguments. Since the interpretation of an operation is different for static and dynamic models, we will give separate definitions for the interpretation of a closed computation in each kind of model.

Definition 8 Interpretation of a Computation in a Static Model

Let M be an exception algebra model, let $F = M.\text{operations}$, let $n = M.\text{arglength}$, let C be a closed computation with respect to the signature of M, and let $\mathbb{I}$ be a sequence. $\mathbb{I}$ is the interpretation of the computation C in the model M if and only if all of the following conditions hold:

1. length($\mathbb{I}$) = length(C),

2. For each i in the range $1 \le i \le$ length(C),
   $\mathbb{I}[i] = F_{\beta}(x_1, \ldots, x_{n(\beta)})$, where $C[i] = \langle$ op : β, args : s $\rangle$,

3. For each j in the range $1 \le j \le n(\beta)$
   x_j = **obj**($\mathbb{I}[k]$) [w], where $s[j] = \langle$ step : k, tc : τ, result : w $\rangle$, and

4. **tc**($\mathbb{I}[k]$) = τ.

A computation is a sequence of operation names and argument specifications, and the interpretation of a computation in a model is the sequence of values obtained by applying the interpretations of the specified operations in the model to data objects specified by the argument specifications. The set of operations of a model is indexed by a set of operation names, and the indexing function specifies the interpretation of each operation name in the model. Since an operation may return more than one data object, the interpretation of a computation is a sequence of tuples of data objects, injected into the component of the disjoint union corresponding to the termination condition produced by the operation. Recall that the range of each operation of an exception algebra is a disjoint union of a set indexed by termination conditions. Each element of a disjoint union is a pair, containing a tag and a data object. If y is the result of some operation of an exception algebra, then **obj**(y) denotes the object without the tag, and **tc**(y) denotes the tag, which is the name of a termination condition.

The interpretation of the computation $C1$ (shown above) in the usual model of pure LISP is the following:

$\mathbb{I}1$ [1] = $\langle$ **normal**, $\langle$ *nil* $\rangle$ $\rangle$
$\mathbb{I}1$ [2] = $\langle$ **normal**, $\langle \langle$ *nil* $\rangle \rangle$
$\mathbb{I}1$ [3] = $\langle$ **normal**, $\langle \langle$ *nil nil* $\rangle \rangle$

The pairs stemming from the disjoint union are shown explicitly. The first component of the

pair is the tag (termination condition), and the second component is the sequence of data objects resulting from each operation. Since all of the operations shown in this example return a single value, the resulting data objects are contained in sequences of length one.

Note that the termination condition of each step must match the termination condition required by every argument specification that uses the results of that step. A closed computation may or may not have an interpretation in a model. If an interpretation exists, it is unique, because the operations of a exception algebra are functions, which necessarily have unique values. A computation may fail to have an interpretation in a given model because the operation specified by some step of the computation may terminate in a different condition than the one required by some later step that uses the results of the given step. If several steps of a computation make conflicting requirements on the termination condition of a given step, then that computation will not have an interpretation in *any* model of the abstraction. If a computation has an interpretation in a model, we will say that the computation is feasible in that model. A feasible computation can involve steps with exceptional termination conditions, and it is possible for the termination condition of the final step to be normal even if the termination conditions of some intermediate steps are exceptional.

The interpretation of a closed computation in a dynamic model is similar, except that there is an extra component containing the system state. Recall that the first argument and the first return value of every operation of a state machine is a system state.

Definition 9 Interpretation of a Computation in a Dynamic Model

Let M be a state machine, let $F = M.\text{operations}$, let $n = M.\text{arglength}$, let C be a closed computation with respect to the signature of M, and let $\mathbb{I}$ be a sequence. $\mathbb{I}$ is the interpretation of the computation C in the model M if and only if all of the following conditions hold:

1. length($\mathbb{I}$) = length(C),

2. For each i in the range $1 \le i \le$ length(C),
   $\mathbb{I}[i] = F_\beta(\sigma_i)\ (x_1, \ldots, x_{n(\beta)})$, where
   $C[i] = \langle$ op : β, args : $s \rangle$,
   $\sigma_i = \lambda\, x$. **undefined** if $i = 1$
   σ_i = **obj**($\mathbb{I}[i-1]$) [1] if $i > 1$

3. For each j in the range $1 \le j \le n(\beta)$
   x_j = **obj**($\mathbb{I}[k]$) [w + 1], where $s[j] = \langle$ step : k, tc : τ, result : $w \rangle$, and

4. **tc**($\mathbb{I}[k]$) = τ.

The initial state for any computation sequence is the empty state, which maps every data object into the initial data state **undefined** and thus has an empty population (i.e., no data objects have been created in the initial state). Each step of a computation except for the first step starts with the state produced by the previous step. The interpretation of a computation in a dynamic model is a sequence of tuples, whose first component is a system state, and whose remaining components are the tuples of data objects and the system states produced by the operations specified by the closed computation. Since the first return value of an operation of a state machine is always a system state, the w-th data object returned by an operation of the abstract type corresponds to the $(w+1)$-st component of the sequence of values returned by the interpretation of the operation in the state machine.

If a computation has an interpretation in a given model, then the value of the computation in that model is the result of the last step of the computation.

Definition 10 Value of a computation

If the computation C has the interpretation $\mathbb{I}$ in the model M, then the value of C in M is **obj**($\mathbb{I}$[length($\mathbb{I}$)]) if M is a static model, and the value of C in M is $\langle v[2], \ldots, v[\text{length}(v)] \rangle$ where v = **obj**($\mathbb{I}$[length($\mathbb{I}$)]) if M is a dynamic model.

Note that the value of a computation can be a tuple containing more than one data object. The

final state of the interpretation of a computation in a state machine is not part of the value, since it is not directly externally observable.

We are now ready to define behavioral equivalence.

Definition 11 Behavioral Equivalence of Models

Two models *M1* and *M2* are behaviorally equivalent if and only if all of the following conditions hold:

1. *M1* and *M2* have the same signature *S*.
2. For any finite closed computation *C* with respect to the signature *S*, *C* has an interpretation in *M1* if and only if it has an interpretation in *M2*.
3. For any finite closed computation *C* with respect to the signature *S*, *C* has an interpretation in *M1* and the value of *C* in *M1* is the boolean value *t* if and only if *C* has an interpretation in *M2* and the value of *C* in *M2* is the same boolean value *t*.

Two models are behaviorally equivalent if they have the same signature, interpretations for the same set of closed computations, and if every computation resulting in a boolean value has the same value in both models.

Theorem 1 : Behavioral equivalence is an equivalence relation.

Proof : The theorem follows directly from the definition.
End of Proof

We intend two models to be behaviorally equivalent if and only if they have the same externally observable behavior. In practice, what we can really observe is the output of a program, which is usually manifested as characters printed on a piece of paper, or displayed on a terminal. Although there is a wide variety of peripheral devices that can be connected to a computer, capable of producing a wide variety of observable effects, they can all be modeled by a (mutable) output stream data abstraction sufficiently well for our purposes, since we are not

concerned with the actual physical properties of the output, but only with whether or not two outputs are distinguishable. We model the data states of an output stream as finite sequences of integers (which can be interpreted as character codes in most cases). We assume output streams have an operation that returns the current state of the stream, represented as an immutable sequence of integers. This operation models the system user, who observes and compares the actual outputs of the system, and it need not actually be implemented. It is included because some data abstractions have properties which can affect the printed output, but which cannot be tested by another program.

Integer sequences are defined to be a priori distinguishable because they are used to model physically observable outputs of the system. Note that the states of mutable data abstractions other than output streams are not a priori observable. We will further assume that integer sequences have an *equal* operation which allows us to reduce the problem of comparing sequences of integers, representing states of output streams, to the much simpler problem of comparing truth values.

The domain of truth values is *a priori* distinguishable because of our assumption that the host programming language provides some means of altering the flow of control depending on a truth value. For example, a conditional statement that prints a different message on each arm can be used to physically distinguish between the truth values. Because of this property of truth values, we insist that the boolean abstraction be given the standard interpretation in all of the models that will enter our discussion. In the standard interpretation, there are exactly two truth values, T and F, with the operations *and*, *or*, *not*, *implies*, and *equivalence* (see Section 4.2.1 and Appendix I).

Different termination conditions are also externally observable, because we can

associate handlers that print different messages with each exception. We do not have to introduce any extra machinery to treat this case, because it is already covered by our definition of the interpretation of a computation. If the final step of a computation C results in two different termination conditions in two different models, then by adding one more step that uses the results of the last step of C and that requires it to terminate in one of the two observed termination conditions, we will get a closed computation C' that is feasible in one model but not in the other.

In our definition of behavioral equivalence, we have assumed that all of the aspects of the behavior of a data abstraction can be observed by means of the operations of the abstraction and its subordinate abstractions. If every operation of every abstraction in the system computes results that depend only on the data objects explicitly passed in as arguments or on the data states in the reachability closure of the arguments (see Section I.3), then this assumption is justified. An example of a system that violates this assumption is the following. Suppose that the abstraction NASTY has an operation *count* that returns a natural number representing the number of objects of type NASTY that have been created so far, and that the only operation that creates new objects of type NASTY is the nullary *create* operation. In order to implement this behavior, the *create* and *count* operations must share some own data. If some other abstraction A in the system is implemented using a representation containing a object of type NASTY, then the operations of A can have effects which are only observable by means of the *count* operation of NASTY, even though NASTY is not necessarily subordinate to A (i.e., A need not have any operations that operate on or return any objects of type NASTY). In general, abstractions with state components that are associated with the type as a whole rather than with any individual data object cannot be used to represent objects of other types without

introducing hidden interactions of the sort described above. Because we want the behavior of a data abstraction to be independent of the representation used in any particular implementation, we exclude structures like NASTY from our discussion. The specification language presented in Chapter 4 has been designed so that abstractions violating this locality assumption cannot be defined.

3.3 Reduced Models

Data abstractions are identified with equivalence classes of models with respect to the behavioral equivalence relation. In this section we will show how to construct a representative element of such an equivalence class, known as a *reduced model*, which can be used to specify the behavior common to all of the members of the class. Reduced models are shown to be unique up to isomorphism, and they are minimal in the sense that they contain no unnecessary elements. Models to be used as specifications for data abstractions should be reduced, since irrelevant components serve no useful purpose and may lead to confusion.

The concept of a reduced model has to be defined somewhat differently for static and for dynamic models. The two cases are discussed below.

3.3.1 Reduced Static Models

Before we can precisely define what we mean by a reduced model, we have to introduce some auxiliary concepts. A reduced model should be free of "extra" objects that cannot influence the externally observable behavior of the model.

Definition 12 Reachable Objects

A data object x is *reachable* in a model M with a signature S if and only if there is some finite closed computation C with respect to S such that x is the value of C in M.

Only the reachable objects in the phyla of a model can influence the externally observable behavior of a model.

We would also like a reduced model not to contain redundant copies of the same object, if there is no observable property that can distinguish between the copies. To arrive at a definition for the external equivalence relation on data objects, we have to define open computations.

Definition 13 Open Computation for a Static Model

An open computation with respect to a signature S and a type $\alpha \in S.\text{typenames}$
is a finite sequence C such that
$C[i] = \langle \text{op} : f, \text{args} : s \rangle$ for each i in the range $2 \le i \le \text{length}(C)$,
where $f \in S.\text{opnames}$, and s is a sequence such that
$\text{length}(s) = S.\text{arglength}(f)$,
$s[j] = \langle \text{step} : n, \text{tc} : \tau, \text{result} : k \rangle$, where
$1 \le n < i$,
if $n = 1$ then $S.\text{argtype}(f, j) = \alpha$; $\tau = \text{normal}$, and $k = 1$,
and if $n > 1$ then $\tau \in S.\text{tc}(C[n].\text{op})$
$1 \le k \le S.\text{rlength}(C[n].\text{op}, \tau)$
and $S.\text{rtype}(C[n].\text{op}, \tau, k) = S.\text{argtype}(f, j)$
for each j in the range $1 \le j \le \text{length}(s)$.

An open computation is just like a closed computation, except that an initial data object is specified, which can be used in any subsequent step of the computation, in addition to the data objects produced by the preceding steps. The initial data object is a parameter to the open computation, and the value of an open computation is a function of this parameter.

Definition 14 Interpretation of an Open Computation in a Static Model

Let M be an exception algebra model, let $F = M.\text{operations}$, let $n = M.\text{arglength}$, let C be a closed computation with respect to the typename α and the signature of M, let $x \in P_\alpha$, and let $\mathbb{I}$ be a sequence. $\mathbb{I}$ is the interpretation of the computation C

applied to the object x in the model M if and only if all of the following conditions hold:

1. length($\mathbb{I}$) = length(C)

2. $\mathbb{I}[1]$ = ⟨ **normal**, ⟨ x ⟩ ⟩

3. For each i in the range $2 \le i \le$ length(C),
   $\mathbb{I}[i] = F_{\beta}(x_1, \ldots, x_{n(\beta)})$, where $C[i]$ = ⟨ op : β, args : s ⟩, and

4. For each j in the range $1 \le j \le n(\beta)$
   x_j = **obj**($\mathbb{I}[k]$) [w], where $s[j]$ = ⟨ step : k, tc : τ, result : w ⟩, and

5. **tc**($\mathbb{I}[k]$) = τ.

The interpretation of an open computation is like the interpretation of a closed computation, except that the interpretation of the first step of the computation is a sequence of length 1, containing the specified initial data object x, and with a **normal** termination condition. We have injected the initial data object x into the **normal** component of a disjoint union for the sake of uniformity. The ⟨ tag, object ⟩ pair is shown explicitly in condition 2.

Definition 15 Value of an Open Computation in a Static Model

If C is an open computation with respect to the type α and the signature S, M is a model with signature S, $x \in M \cdot \text{phyla}_{\alpha}$, and if $\mathbb{I}$ is the interpretation of C in M with respect to α, then the value of C applied to x in M is $C(x)$ = **obj**($\mathbb{I}$[length($\mathbb{I}$)]).

The value of an open computation is the tuple of data objects resulting from the last step of the computation when interpreted in the given model.

Definition 16 External Equivalence of Objects in a Static Model

Let M be a model, $\alpha \in M \cdot$ typenames, and x1, x2 $\in M \cdot \text{phyla}_{\alpha}$. The the data objects x1 and x2 are externally equivalent if and only if for every open computation C with respect to α all of the following conditions hold:

1. C has an interpretation in M with respect to the data object x1 if and only if C has an interpretation in M with respect to the data object x2.

2. C has an interpretation in M with respect to $x1$ and the value of C applied to $x1$ in M is the boolean value t if and only if C has an interpretation in M with respect to $x1$ and the value of C applied to $x2$ in M is the same boolean value t.

Two objects of a given model are externally equivalent if and only if every open computation applied to one of the objects yields a result that is indistinguishable from the result of applying the same open computation to the other object. This means that the two objects share all externally observable properties, and therefore represent the same abstract object, even if they are two distinct objects in the model. The point is that the identities of the objects in a model are not externally observable unless the data abstraction provides some operations that make them observable.

Now we are ready to define reduced static models.

Definition 17 Reduced Static Model

A static model M is *reduced* if and only if all of the following conditions hold:

1. For each $\alpha \in M.\text{typenames}$ and for each $x \in M.\text{phyla}_{\alpha}$, x is reachable.
2. For each $\alpha \in M.\text{typenames}$ and for each $x1, x2 \in M.\text{phyla}_{\alpha}$, if $x1$ and $x2$ are externally equivalent, then $x1 = x2$.

A reduced static model has no extra objects, since every object is the result of some finite closed computation, and hence externally observable, and every distinct pair of objects in the model differs in some externally observable property.

Theorem 2 : Every equivalence class of models with respect to the behavioral equivalence relation contains a reduced model.

Proof : Take the reachable subset, and divide by the external equivalence relation. Details in Appendix III.
End of Proof

Theorem 3 : If two reduced models are behaviorally equivalent, then they are isomorphic.

Proof : The isomorphism maps the value of every closed computation in one model into the value of the same computation in the other model. Details in Appendix III.
End of Proof

Thus every constant data abstraction has a reduced model that is unique up to isomorphism.

Theorem 4 : If *M* is behaviorally equivalent to *M'* and *M* is reduced, then there is a homomorphism from a subset of *M'* onto *M*.

Proof : The construction of theorem 2 yields a reduced model which is a homomorphic image of M. Compose that homomorphism with the isomorphism guaranteed by theorem 3. Details in Appendix III.
End of Proof

We can always find a homomorphism from an arbitrary static model to a behaviorally equivalent reduced model. This result is interesting because the classical way to prove the correctness of an implementation of a data abstraction with respect to an abstract model specification is to construct such a homomorphism from the implementation to the defining model. The theorem says that the required homomorphism exists for any correct static implementation model, provided that the defining model is reduced. While there is no guarantee that the homomorphism is computable or even finitely describable, the homomorphisms corresponding to most implementations are quite tractable. As we shall see in the next subsection, the corresponding theorem for dynamic models is false.

3.3.2 Reduced Dynamic Models

Informally, a model is reduced if it has no unnecessary objects. We have to take a different approach to formalizing this concept for dynamic models, because the existence of a data object and the properties of a data object are not completely determined by the identity of

the object, since they will in general depend on the system state. Theorem 4 fails for dynamic models for this very reason. A homomorphism on a many sorted algebra is a family of mappings, one for each phylum. In a dynamic model, the elements of any phylum corresponding to the principal type of a dynamic data abstraction have no distinguishing properties except for their identity. All of the interesting properties of an object belonging to such a phylum come from the image of that object under the system state function, and any particular object does not have any interesting properties until it is created (i.e., until some operation gives the object a data state other than **undefined**). Depending on how an object gets created in each particular computation, an object in the model can come to represent any of a number of different abstract objects. Consequently, there may be no correspondence between the objects of one model and those of another which is both consistent with the operations and independent of the computation history. The cases where the correspondence is independent of the computation history are rare.

The rest of this section consists of a characterization of a reduced dynamic model, and an example of two behaviorally equivalent models such that one is reduced but is not a homomorphic image of the other.

There are two requirements a dynamic model must meet if it is to be reduced: the phyla must contain no unnecessary objects, and for every state, the population must contain no unnecessary objects. If we insist that every element of every phylum must be reachable, the first requirement is met. Reachability can be defined for dynamic models in a way entirely analogous to the definition for static models, and presents no essential difficulty. For most dynamic models a countable infinity of data objects are reachable, and each data object has no directly observable properties except for its identity, so that the first requirement is not very

interesting. The second requirement requires a fundamentally new approach, because there is no way to meaningfully define the behavior of an abstract object independently of the system state.

We will assume that an operation of a data abstraction can create at most finitely many new data objects [cf. 15]. Since we require all operations to terminate in a finite amount of time, and since all real machines compute at a finite rate, this assumption is justified. A consequence of this assumption is that the population of every reachable state is finite, where a state is reachable if and only if there is some finite closed computation that produces that state.

We can define reduced models for dynamic models as follows.

Definition 18 Reduced Dynamic Model

A dynamic model M is *reduced* if and only if there is no other model M' such that M' is behaviorally equivalent to M, and for some closed computation C, the cardinality of the population of the final state produced by C in M' is strictly smaller than the cardinality of the population of the final state produced by C in M.

An example of a case where we have a reduced dynamic model $M1$, and a behaviorally equivalent model $M2$ such that there is no homomorphism from any subset of $M2$ onto $M1$ is described below.

Consider a version of mutable lists, which have *nil* as the only atom, and for which the *rplaca* and *rplacd* operations return the list that was modified rather than the old value of the component that was replaced, as is the case in LISP. The model $M1$ has $P_{list} = \mathbf{N}$, and $\Delta_{list} = \{ nil \} \sqcup (\mathbf{N} \times \mathbf{N})$. In $M1$, the only operation that extends the population of the list domain is *cons*. The *eq* operation serves to make the identity relation on objects in the model externally observable, so that every newly created object is distinguishable from any previously existing object, and hence $M1$ is reduced.

The model $M2$ has $P_{list} = \mathbf{N}$ and $\Delta_{list} = cell[\{ nil \} \cup (\mathbf{N} \times \mathbf{N})]$. In $M2$, *rplaca* and *rplacd* as well as *cons* extend the population of P_{list}. We have introduced an extra level of indirection, so that the identities of the abstract objects correspond to the identities of the cells that are the data states of the elements of P_{list}, rather than to the elements of P_{list} directly, as would be the case for any reduced model. $M2$ is behaviorally equivalent to $M1$, but $M2$ is not reduced, because the *rplaca* and *rplacd* operations create redundant list objects.

There can be no homomorphism from $M2$ to $M1$ because the correspondence between objects in $M2$ and objects in $M1$ depends on the system state. For example, the computation shown in $C1$ below

$C1[1] = \langle$op : *nil* , args : $\langle\rangle\rangle$
$C1[2] = \langle$op : *cons* , args : $\langle\langle$step : 1, tc : normal, result : 1$\rangle$, $\langle$step : 1, tc : normal, result : 1$\rangle\rangle\rangle$
$C1[3] = \langle$op : *cons* , args : $\langle\langle$step : 1, tc : normal, result : 1$\rangle$, $\langle$step : 1, tc : normal, result : 1$\rangle\rangle\rangle$

has the following interpretation in $M1$:

$\mathbf{I}_{11}[1] = \langle \sigma_0 , 0 \rangle$ where $\sigma_0(0) = nil$
$\mathbf{I}_{11}[2] = \langle \sigma_1 , 1 \rangle$ where $\sigma_1(0) = nil$, and $\sigma_1(1) = \langle 0, 0 \rangle$
$\mathbf{I}_{11}[3] = \langle \sigma_2 , 2 \rangle$ where $\sigma_2(0) = nil$, $\sigma_2(1) = \langle 0, 0 \rangle$, and $\sigma_2(2) = \langle 0, 0 \rangle$

$C1$ evaluates the expression "(*cons nil nil*)" twice, resulting in two copies of the list (*nil*). Each element of the interpretation $\mathbf{I}_{11}$ is a pair containing the results returned by the operation specified by the corresponding step of the computation $C1$. The first element of each pair is a system state function, and the second component of each pair is a natural number representing a mutable list. Note that the system state is considered to be result 0, and that result 1 is the first data object returned by the operation, corresponding to the second element of each pair.

The computation $C1$ has the following interpretation in $M2$:

$\Pi 12[1] = \langle \sigma_0 , 0 \rangle$ where $\sigma_0(0)$ = cell-0,
σ_0(cell-0) = *nil*

$\Pi 12[2] = \langle \sigma_1 , 1 \rangle$ where $\sigma_1(0)$ = cell-0, $\sigma_1(1)$ = cell-1,
σ_1(cell-0) = *nil*, σ_1(cell-1) = $\langle 0, 0 \rangle$

$\Pi 12[3] = \langle \sigma_2 , 2 \rangle$ where $\sigma_2(0)$ = cell-0, $\sigma_2(1)$ = cell-1, $\sigma_2(2)$ = cell-2
σ_2(cell-0) = *nil*, σ_2(cell-1) = $\langle 0, 0 \rangle$, σ_2(cell-2) = $\langle 0, 0 \rangle$

In model *M*2 we have an extra level of indirection. If the state σ_{M1} of *M*1 corresponds to the state σ_{M2} of *M*2, then we have the relation $\sigma_{M1}(n) = \sigma_{M2}(\sigma_{M2}(n))$ for any $n \in \mathbf{N}$ (a natural number representing a mutable list). The correspondence between the elements of P_{list} for the final state produced by *C*1 in *M*2 and the elements of the population of P_{list} in the final state produced by the interpretation of *C*1 in *M*1 is

*M*2		*M*1
0	→	0
1	→	1
2	→	2

Now consider the computation *C*2 shown below.

*C*2[1] = ⟨op : *nil* , args : ⟨⟩⟩
*C*2[2] = ⟨op : *cons* , args : ⟨⟨step : 1, tc : **normal**, result : 1⟩, ⟨step : 1, tc : **normal**, result : 1⟩⟩⟩
*C*2[3] = ⟨op : *rplaca* , args : ⟨⟨step : 2, tc : **normal**, result : 1⟩, ⟨step : 1, tc : **normal**, result : 1⟩⟩⟩

*C*2 computes the expression "(*rplaca* (*cons nil nil*) *nil*)". The interpretation of *C*2 in *M*1 is

$\Pi 21[1] = \langle \sigma_0 , 0 \rangle$ where $\sigma_0(0)$ = *nil*.
$\Pi 21[2] = \langle \sigma_1 , 1 \rangle$ where $\sigma_1(0)$ = *nil*, and $\sigma_1(1) = \langle 0, 0 \rangle$.
$\Pi 21[3] = \langle \sigma_1 , 2 \rangle$ where $\sigma_2(0)$ = *nil*, and $\sigma_2(1) = \langle 0, 0 \rangle$.

The interpretation of *C*2 in *M*2 is

$\mathbf{I}22[1] = \langle \sigma_0 , 0 \rangle$ where $\sigma_0(0)$ = cell-0, and
σ_0(cell-0) = *nil*.

$\mathbf{I}22[2] = \langle \sigma_1 , 1 \rangle$ where $\sigma_1(0)$ = cell-0, $\sigma_1(1)$ = cell-1,
σ_1(cell-0) = *nil*, and σ_1(cell-1) = $\langle 0, 0 \rangle$.

$\mathbf{I}22[3] = \langle \sigma_2 , 2 \rangle$ where $\sigma_2(0)$ = cell-0, $\sigma_2(1)$ = cell-1, $\sigma_2(2)$ = cell-1
σ_2(cell-0) = *nil*, and σ_2(cell-1) = $\langle 0, 0 \rangle$.

Thus the correspondence between the elements of the population of P_{list} in $M2$ and the elements of P_{list} in $M1$ required in the final state produced by $C2$ is

$M2$		$M1$
0	→	0
1	→	1
2	→	1

A homomorphism must be a function, and hence single valued. Since the computations $C1$ and $C2$ introduce conflicting requirements for the image of the element $2 \in P_{list}$, there can be no homomorphism from $M2$ to $M1$.

This example demonstrates that there are some correct implementations whose correctness cannot be established by exhibiting a homomorphism from the implementation to the defining model, even if the defining model is reduced. Therefore other methods of proof relying more directly on the underlying concept of behavioral equivalence are needed. Proofs of correctness of implementation are discussed in Chapter 5.

4. Specification Language

In Chapter 3 we saw how a data abstraction could be identified with an equivalence class of models with respect to the behavioral equivalence relation. It is our thesis that an effective and useful technique for specifying a data abstraction is to explicitly construct a (reduced) model of the abstraction. The data abstraction denoted by such a specification is the class of all models behaviorally equivalent to the model that was constructed, which will be referred to as the standard model. In order to define a standard model for a data abstraction, we must specify the signature of the data abstraction, and give interpretations for its phyla and operations. In this chapter we present a number of methods for doing this, along with a language for describing particular models defined using these methods. Chapter 5 is concerned with proving that a proposed implementation is correct with respect to a given standard model.

Since we are primarily interested in using our specification language for defining particular models, rather than for proving meta-theorems about the specification language, we have made no effort to keep the language minimal. Our intent was to make it easy for people to read and write specifications in our language. Such a goal has no objective measure, and the reader is urged to consider our examples and to construct additional ones in order to judge the merits of the formalism. The syntax and abbreviations we have chosen are meant to ease the task of the human reader. For applications where mechanical processing of the specifications is to play a dominant role, a more restricted syntactic form may be appropriate.

As mentioned in Section 3.1, we will construct models for data abstractions incrementally, assuming at each stage that models for all of the subordinate abstractions have already been defined. We will explicitly construct the interpretation of the principal type, and

implicitly specify that the interpretation of each subordinate type is the principal type of the standard model for its defining abstraction.

In this Chapter we will assume that a model for a static data abstraction is an exception algebra, and that a model for a data abstraction with time dependent behavior is a state machine. (Recall that a state machine is an exception algebra with a distinguished phylum of system states.)

4.1 Components of a Specification

The important part of the specification language is its structure and semantics, which are explained informally below. A precise definition of our somewhat arbitrarily chosen syntax can be found in Appendix IV.

The basic components of a model specification are illustrated by the example shown in Figure 3. This example gives a definition of immutable stacks (or stack states), modeled in terms of sequences, where the top element of the stack is the last element of the sequence representing the stack. This example has been treated many times in the literature on specifications for data abstractions, and will probably be familiar to the reader. Later we will see a specification of mutable stacks. The form of a specification and the meaning of the components are explained briefly below, with occasional reference to the stack example.

The name of the abstraction, which is the same as the name of the principal type, is introduced by the keyword type. An optional abbreviation for the name of the principal type is introduced by the keyword as. The name of the type is followed by an optional list of parameters, enclosed in square brackets. If there is a parameter list, then the specification is not a single definition, but rather a definition schema, which can be instantiated by substituting a

Figure 3. Stack

```
type stack[E]      as S
requires           E : type

with               empty:      → S                                  % the empty stack
                   push:       E × S → S
                   pop:        S → S + ⟨ stack_underflow : ⟩
                   top:        S → E + ⟨ stack_underflow : ⟩
                   null:       S → boolean                          % is s empty?

representation:    sequence[E]
restrictions:      none
identity:          sequence[E]$equal

operations:        empty() = sequence[E]$empty()
                   push(e, s) = addlast(s, e)
                   pop(s) = if *s = 0 then ⟨ stack_underflow : ⟩     % * is length
                                      else s[ 1 .. (*s)-1 ]         % s[ a .. b ] is subrange
                   top(s) = sequence[E]$last(s)
                   null(s) = if *s=0 then true else false
end stack
```

---

suitable expression for the occurrences of each parameter in the body of the definition. If there is a parameter list, there must also be a **requires** clause which specifies the restrictions on the expressions that may be substituted for each parameter. In the stack example, the parameter E is restricted to range over the names of types (E is the name of the type of the elements on the stack).

The keyword **with** introduces a specification of the signature, in the notation introduced in Section 2.3.3. The signature gives the name and type of each externally available operation, including the number and types of arguments and the number and types of return values for each possible termination condition. The set of subordinate types is also implicitly specified, since it contains precisely those types, other than the principal type, that are used as a

component of the domain or range of some operation in the signature. Each operation may also have an alternate syntactic form, which is introduced by the keyword **as**. Within the definition of an alternate syntactic form, the expression "**arg** *n*" stands for the *n*-th argument to the operation, and all of the other symbols (up to the end of the line) are separators (prefix, infix, postfix, etc.), which are to be taken literally. The type of an operation (i.e., the name of its defining abstraction) should be obvious from its context. In cases where it is not obvious, or where we want to emphasize the type, we will use the standard functional notation, where the name of the operation is prefixed by the name of its defining abstraction followed by a "**$**". The parameters of the type will be included in cases where that is helpful to the (human) reader.

The interpretation of the principal type is specified by the next three components. The underlying representation algebra is specified by an expression introduced by the keyword **representation**. The allowable expressions and their meanings are discussed below in Section 4.3 below. The **restrictions** component specifies a subset of the principal type of the representation algebra, and the **identity** section specifies an equivalence relation on that subset. The interpretation of the principal type is the quotient of the specified subset of the principal type of the representation algebra with respect to the specified equivalence relation. The **identity** relation in effect determines the identity of the abstract objects of the principal type of the abstraction being defined, and serves as the logical equality relation for the principal type of the model. Logical equality is not externally available unless one of the operations of the abstraction happens to coincide with it. In a reduced model, logical equality should be externally observable in terms of the operations, although not necessarily in terms of the same finite computation for all objects in its domain.

The operations are defined in a section introduced by the keyword **operations**. The forms and meanings of the operation definitions are described in Section 4.2 below.

Comments can appear at any point in a specification. They are introduced by the symbol "%" and extend to the end of the line.

Auxiliary functions or abbreviations may be used in the definition of the operations. The types of any auxiliary functions must be given in the **internal** section, and the definitions of any auxiliary functions or abbreviations must be given in the **definition** section, in the same form as the types and definitions of the operations. Auxiliary functions are introduced solely for clarity and expressive power, and they are not externally available (for use by programs) or even part of the model, which contains only the functions acting as the interpretations for the externally available operations. Auxiliary functions may be used in assertions and in proofs of properties of the data abstraction.

A specification is terminated by the keyword **end**, optionally followed by the name of the abstraction that was defined. In cases where several data abstractions are **subordinate** to each other it is necessary to define a group of related abstractions by a single model with several principal types. In the specification language, a module defining a model with several principal types consists of the keyword **module**, followed by any number of type definitions, followed by **end module**. The representation and the internal functions of each type are accessible throughout the module. Modules may not be nested.

4.2 Defining Operations

The principal type of a model is the quotient of the subset of the principal type of the representation algebra satisfying the constraints specified in the **restrictions** section with respect to the equivalence relation specified in the **identity** section. If there is no **restrictions** section, the entire principal type is used. If there is no **identity** section, then the logical equality of the principal type of the representation algebra is used, and the quotient structure is trivial, since all of the equivalence classes are singletons in this case.

The definitions of the operations in a type definition in our specification language explicitly define functions that operate on the elements of the principal type of the representation algebra. These functions are implicitly extended to operate on the equivalence classes that make up the principal type of the quotient structure in the usual way, described in more detail in Section 4.4.4.

The following subsections describe the means for defining functions provided in the specification language, and then examine the constraints a function definition has to satisfy in order for it to denote a well formed operation for the exception algebra or state machine being defined.

4.2.1 Conditional Expressions

We will use a language for defining functions similar to that introduced by McCarthy in [33], extended by the iota expressions described in the next subsection.

A function definition consists of a function name, a list of variables, an equals sign, and an expression. Valid expressions are variables, iota expressions, functions applied to

expressions, and conditionals applied to expressions. Conditionals are written with the usual if-then-else syntax, and they have the usual meaning:

$b \Longrightarrow$ ((if b then x else y) = x)
$\neg\, b \Longrightarrow$ ((if b then x else y) = y).

The variables that may appear consist of the variables appearing in the list of formal arguments on the left side of the equals sign, and any local variables defined immediately after the function definition. A local variable is defined by writing its name, an equals sign, and an expression. Circular definitions are not allowed: it must be possible to eliminate all of the local variables from the right hand side of a function definition by a finite number of substitutions, each of which replaces an occurrence of a local variable by the expression defining it. Local variables are a notational convenience, in the sense that any definition using local variables has an equivalent definition without local variables. The abbreviations introduced by local variables can be a very important aid in making the structure of a function definition more apparent to the human reader, and they can at times dramatically shorten the text of a function definition.

The functions that may appear on the left hand side of an operation definition are the primitive operations of the representation algebra and of its subordinate abstractions, and the operations and auxiliary functions defined in the type specification or module in which the defining expression appears. Recursive definitions are allowed. Auxiliary functions must be defined in the definition section. Auxiliary functions can increase the expressive power of the language, as proved for equational axiomatic definitions in [52]. This result should not be surprising, since auxiliary functions may be defined recursively, so that the process of

substituting the body of the function definitions for each invocation (attempting to eliminate the auxiliary functions from the main definition) may fail to terminate.

Since the operations of a data abstraction are supposed to be total functions, it is necessary to show that all recursive definitions used are well founded.

4.2.2 Iota Expressions

Iota expressions are named for the iota operator in logic. An iota expression has the form $x : p(x)$, where x is the only free variable in the predicate $p(x)$. If x is of type T, and if the set $\{ x \in T \mid p(x) \}$ is a singleton set, then the value of the iota expression $x : p(x)$ is the unique element of that set, and otherwise the iota expression is undefined.

Iota expressions are useful in cases where it is much easier to specify a property the result of a function must satisfy and to prove that the property uniquely determines the result than it is to provide a recursive definition of the function. Iota expressions are the equivalent of Hoare style input/output predicates for a language with functions and without side effects. An examples of a definition where an iota expression definition is appropriate is

$$\text{isqrt}(n) = y : y^2 \leq n < (y+1)^2$$

which defines the integer square root function.

It is necessary to show that each iota expression used in a specification is well defined, given the context in which it appears. More precisely, the following two requirements must be satisfied for each iota expression $x : p(x)$.

1. $q(x) \Longrightarrow \exists x\ [\ p(x)\ ]$

2. $\forall x,y\ [\ q(x)\ \&\ p(x)\ \&\ q(y)\ \&\ p(y) \Longrightarrow x \equiv y\ ]$

where $\equiv$ is the equivalence relation defined in the identity section, or the logical equality relation if there is no identity section, x ranges over the principal type, and where $q(x)$ is the path predicate describing the conditions under which the iota expression can get evaluated. Let α be an occurrence of an iota expression in the expression e, and let path(α, e) denote the path predicate for α in e. Then path(α, e) is defined as follows:

path(α, α) = true
if e is "$f(x_1, \ldots, x_n)$" and α occurs in x_i then path(α, e) = path(α, x_i)
if e is "if b then x else y" and α occurs in b then path(α, e) = path(α, b)
if e is "if b then x else y" and α occurs in x then path(α, e) = b & path(α, x)
if e is "if b then x else y" and α occurs in y then path(α, e) = $\neg$ b & path(α, y)

4.3 Constructing Algebras

Our approach will be to define a standard model for a data abstraction in terms of a given representation algebra. The principal type of the standard model will in general be the quotient of a specified subset of the principal type of the representation algebra with respect to a specified equivalence relation. The operations of the standard model will be defined in terms of the operations of the representation algebra, as described in the previous section. The rest of this section is devoted to defining a rich set of representation algebras that can be used as building blocks for defining models.

Since it is not our aim in the present work to investigate the foundations of mathematics, we will assume that logic, truth values, sets, cartesian products, natural numbers and integers are primitive. An excellent formalization of these structures can be found in [48].

We will use the notations summarized below.

T and F denote the truth values *true* and *false* respectively. These are the only truth values, and they are distinct. &, $\vee$, $\neg$, $\Longrightarrow$, and $\equiv$ denote the *and*, *or*, *not*, *implies*, and *equivalence* operations on the truth values, respectively, and $\forall$ and $\exists$ denote the universal and existential quantifiers. $\in$, $\cup$, $\cap$, and - denote set membership, union, intersection, and set difference. Finite sets are written $\{ x_1 , \ldots , x_n \}$ and finite cartesian products or n-tuples are written $\langle x_1 , \ldots , x_n \rangle$. The i-th component of an ntuple X is written $X \downarrow i$, so that $\langle x_1 , \ldots , x_n \rangle \downarrow i = x_i$. The set of natural numbers is denoted by N. 0, σ, $+$, $*$, $<$, and $=$ denote zero, successor, plus, times, less than, and logical equality on N, respectively. The set of integers is denoted by Z, and $+$, $*$, $-$, *quotient*, *remainder*, *abs*, $<$, and $=$ denote plus, times, (unary) minus or (binary) subtraction, the quotient and the remainder of integer division, the absolute value operation, the less than relation, and the equals relation, respectively. We rely on the context to differentiate between operations on the integers and operations on the natural numbers with the same name. The usual decimal notation will be used for integer constants, which are considered to be an infinite class of nullary operations from the formal point of view.

We will define a number of ways for defining algebras, namely finite enumerations, finite cartesian products, finite disjoint unions, finite power sets, finite sequences, and recursive definitions (fixpoint equations). The set of representation algebras is defined to be the set generated by the standard model for the boolean algebra defined below with respect to the constructions listed above (i.e., the smallest set of algebras that is closed with respect to the constructions for generating new algebras). Each of the constructions supplies a set of operations as well as a set of data objects, so that we are generating a set of algebras rather than merely a set of sets.

We also define two special purpose algebras, **token** and **state[D]**, for use in defining the phylum of system states in a state machine model. Tokens and states have interdependent meanings, and are defined by a single module with two principal types. These two abstractions codify the ways in which the operations of a state machine can depend on the system state.

4.3.1 Booleans

We want to have an image of the domain of truth values as one of our representation algebras. Since everything else depends on the boolean domain (predicate operations return values of type boolean), we cannot use the methods described below to define it without introducing a circularity. We will define booleans in terms of the truth values in the underlying mathematics. A necessarily informal definition in a notation similar to our specification language is shown in Figure 4. Because the meaning of a data abstraction is defined in terms of booleans (cf. behavioral equivalence, Chapter 3), we insist that the booleans be given their standard interpretation in all models under discussion.

Note that the *equal* operation on the booleans is the same as logical equivalence on the underlying domain of truth values, which in turn is the same as the logical equality on the boolean domain. In keeping with our policy that the only externally observable properties of a data abstraction are those that can be calculated in terms of the operations, we will always interpret "=" as the *equal* operation of the defining abstraction of the type of the data objects being compared. Thus it is proper to use "=" in the definition of an operation only if the representation type has an *equal* operation. Care must be taken that the *equal* operation of all

Figure 4. Boolean Abstraction

type boolean	**as B**		
with	true:	$\rightarrow B$	
	false:	$\rightarrow B$	
	not:	$B \rightarrow B$	**as ¬ arg 1**
	and:	$B \times B \rightarrow B$	**as arg 1 & arg 2**
	or:	$B \times B \rightarrow B$	**as arg 1 ∨ arg 2**
	implies:	$B \times B \rightarrow B$	**as arg 1 ⟹ arg 2**
	equal:	$B \times B \rightarrow B$	**as arg 1 = arg 2**
representation	B = truth values		
operations	true() = T		
	false() = F		
	not(x) = if x then F else T		
	and(x, y) = if x then y else F		
	or(x, y) = if x then T else y		
	implies(x, y) = (¬x) ∨ y		
	equal(x, y) = (x ⟹ y) & (y ⟹ x)		
end boolean			

the algebras defined is in fact an identity relation.[1] Logical equality is assumed to be defined for the structures that have been imported from the underlying mathematics, such as the natural numbers.

The boolean type is isomorphic to the domain of truth values in the underlying logic, as indicated by the interpretations of the operations *true* and *false* in the standard model for the booleans. The operations of the representation algebra correspond to those of the propositional calculus in the underlying logic. Quantifiers are defined only in the underlying logic, and have no counterpart in the representation algebra. We will make heavy and implicit

1. An identity relation *equal* must be reflexive, symmetric, transitive, and must satisfy the substitution property $equal(x, y) \Rightarrow P(x) \equiv P(y)$, for any predicate P.

use of the isomorphism between the booleans and the underlying domain of truth values, so that the primitive predicates of any representation algebra, which return values of type boolean, can be combined with quantifiers, and used in if-then-else expressions, both of which are defined in terms of the underlying logic. The booleans are the only type for which we will talk about properties of the interpretations of the objects directly. For all other types, we will talk only about the results of applying the primitive operations. The only direct connection to the underlying mathematics is by means of the booleans, which is why that type is given a distinguished status.

4.3.2 Natural Numbers and Integers

We import the systems of integers and natural numbers directly from the underlying mathematics. Definitions of these types are given in Appendix II. These definitions serve to pin down the syntax, and have nothing surprising in them.

4.3.3 Enumerations

Enumerations are useful for defining small finite sets, such as characters. Larger finite sets, such as fixed length integers, are most conveniently described in terms of the infinite sets they are intended to approximate, as will be illustrated later in this chapter.

An enumeration $\{ x_1 , \ldots , x_n \}$ defines an algebra whose principal type is a set with n elements, and whose only subordinate type is *boolean*. The algebra has n nullary operations, the constants x_i for $1 \leq i \leq n$, and one binary operation, *equal*, which allows the elements of the principal type to be distinguished from each other. We want equal(x_i, x_j) to be true if and only if $i = j$. The indices range over the set of natural numbers **N**. There are many models that

Figure 5. Enumeration Types

type { x_1 , ... , x_n }		**as** T	
with	x_i:	→ T	for $1 \le i \le n$
	equal:	T x T → boolean	
representation:	natural numbers		
restrictions:	i such that $1 \le i \le n$		
identity:	=		
operations:	$x_i() = i$		for $1 \le i \le n$
	equal(a, b) = if a = b then true else false		
end			

---

exhibit the behavior described above. Our standard model, shown in Figure 5, uses natural numbers to represent the elements of the enumeration. The "=" operation used in defining the *equal* operation of the enumeration type denotes the equality operation of the natural numbers.

4.3.4 Tuples

Tuples are labeled finite cartesian products. We will write **tuple**$[w_1 : S_1 , \ldots , w_n : S_n]$ for the set of n-tuples such that the i-th component is a member of the set S_i and bears the label w_i, for each i in the range $1 \le i \le n$. We will write $\langle w_1 : x_1 , \ldots , w_n : x_n \rangle$ for the tuple containing the elements $x_1 , \ldots , x_n$. The projection function mapping a tuple to its i-th component is denote by $p[w_i]$, and if t is a tuple, then $p[w_i](t)$ can be abbreviated as $t \cdot w_i$. If $t = \langle w_1 : x_1 , \ldots , w_n : x_n \rangle$, then $t \cdot w_i = x_i$ for each i in the range $1 \le i \le n$. Two tuples are equal if and only if corresponding components are equal. Equality of tuples is defined for the type **tuple**$[w_1 : S_1 , \ldots , w_n : S_n]$ if and only if the defining algebra of S_i has an *equal* operation for each i in the range $1 \le i \le n$ which is an identity relation. If some of the component types

Figure 6. Tuple

type	tuple[$w_1 : S_1 , \ldots , w_n : S_n$]		**as** T
requires	S_i : type		for $1 \le i \le n$
with	construct:	$S_1 \times \ldots \times S_n \rightarrow T$	**as** $\langle w_1 : x_1 , \ldots , w_n : x_n \rangle$
	$p[w_i]$:	$T \rightarrow S_i$	**as arg** 1. w_i for $1 \le i \le n$
	equal:	$T \times T \rightarrow$ boolean	
representation	$T = S_1 \times \ldots \times S_n$		
restrictions	none		
identity	equal		
operations	construct($x_1, \ldots , x_n$) = $\langle x_1 , \ldots , x_n \rangle$		
	$p[w_i](x) = x \downarrow i$		
	equal(x, y) = if $\forall i$ [$1 \le i \le n \Rightarrow x.w_i = y.w_i$] then true else false		
end tuple			

do not have equality operations, then the tuple type does not have an *equal* operation either, although the type and all of the other operations on it are well defined.[2]

This description is summarized in Figure 6 in an informal notation. Recall that cartesian products are primitive, and that if x is an n-tuple, then $x \downarrow i$ denotes the i-th component of the n-tuple.

2. An *equal* operation will be defined for every representation algebra in our basic set. It is also possible to construct tuples with components from user defined types, which need not have an *equal* operation (e.g. stacks).

4.3.5 Oneofs

Oneofs are finite labeled disjoint unions. A oneof is the dual of a tuple, in the sense that the projection functions go in the other direction. We will write oneof$[w_1 : S_1, \ldots, w_n : S_n]$ for the disjoint union of the sets $S_1, \ldots, S_n$. Our standard model for oneof$[w_1 : S_1, \ldots, w_n : S_n]$, shown in Figure 7, uses the set $\bigcup_{1 \le i \le n} \{w_i\} \times S_i$, to represent the principal type, which coincides with the standard interpretation for disjoint unions used in classical mathematics. Each element of a disjoint union is represented as a pair containing an element of one of the component types, and a label indicating which component the element came from. If an element occurs in more than one of the S_i, it will occur in several distinct elements of the disjoint union, distinguished by different values for the label component of the pair.

Figure 7. Oneof

type oneof$[w_1 : S_1, \ldots, w_n : S_n]$ **as** O

requires	S_i : type			for $1 \le i \le n$
with	in$[w_i]$:	$S_i \rightarrow$ O	**as arg** 1 **in** w_i	for $1 \le i \le n$
	to$[w_i]$:	O $\rightarrow S_i +$ ⟨ wrong_type : ⟩	**as arg** 1 **to** w_i	for $1 \le i \le n$
	is$[w_i]$:	$S_i \rightarrow$ boolean	**as arg** 1 **is** w_i	for $1 \le i \le n$
	equal:	O × O → boolean		
representation	O = $\bigcup_{1 \le i \le n} \{ w_i \} \times S_i$			
restrictions	none			
identity	=			
operations	in$[w_i]$(x) = ⟨ w_i, x ⟩			
	to$[w_i]$(o) = if o ↓ 1 = w_i then o ↓ 2 else ⟨ wrong_type : ⟩			
	is$[w_i]$(o) = if o ↓ 1 = w_i then true else false			
	equal(o1, o2) = if (o1 ↓ 1 = o2 ↓ 1 & o1 ↓ 2 = o2 ↓ 2) then true else false			
end oneof				

A **oneof** type has n injections from the component types into the disjoint union, n predicates indicating whether or not an element of the disjoint union came from a given component, and n projections, which return the element without the label if the label corresponds to the component of the projection, and which terminate in the *wrong_type* exception with no return value otherwise. The oneof type has an *equal* operation if and only if each component type has an *equal* operation.

As we shall see below, one of the main uses for disjoint unions is in constructing recursively defined types, such as trees.

4.3.6 Sets

We will write **set**[E] for the domain of finite subsets of the type E. An informal definition of **set**[E] is shown in Figure 8. This construction is valid only if the defining abstraction of the type E has an *equal* operation that computes an identity relation, because equality is necessary for deciding set membership. There is one nullary operation which returns the empty set of the given type, and there are operations for adding and removing elements, and for forming unions, intersections, set differences, and restrictions. There are also operations for testing to see if an element belongs to a given set, if one set is a subset of another, if two sets have the same members, and for finding the size of a set, which is always defined because we are dealing only with finite sets. Set restriction is treated as an indefinitely large parameterized family of operations, where the parameters are the bound variable and the body of a lambda expression defining a predicate (i.e., a function from E to boolean). The size of a set is defined to be an integer rather than a natural number, so that sizes can be subtracted and divided. The natural numbers and the integers are defined in Appendix II.

Figure 8. Set

type set[E] as S
requires E : type with E$equal: E × E → boolean such that ident_op(E$equal)

with	null:	→ S	
	add:	E × S → S	
	remove:	E × S → S	
	union:	S × S → S	as arg 1 ∪ arg 2
	intersection:	S × S → S	as arg 1 ∩ arg 2
	difference:	S × S → S	as arg 1 - arg 2
	restrict[x, p(x)]:	S → S	as { x : arg 1 | p(x) }
	empty:	S → boolean	
	member:	E × S → boolean	as arg 1 ∈ arg 2
	subset:	S × S → boolean	as arg 1 ⊆ arg 2
	equal:	S × S → boolean	as arg 1 = arg 2
	size:	S → int	as | arg 1 |

representation S = mathematical sets
restrictions s such that $s \subseteq E$ and cardinality(s) ∈ N
identity equal

operations
null() = { }
add(e, s) = s ∪ { e }
remove(e, s) = s - { e }
union(s1, s2) = s1 ∪ s2
intersection(s1, s2) = s1 ∩ s2
difference(s1, s2) = s1 - s2
restrict[x, p(x)](s) = { x ∈ s | p(x) }
empty(s) = if ∃ x [x ∈ s] then false else true
member(e, s) = if e ∈ s then true else false
subset(s1, s2) = if ∃ x [x ∈ s1 & ¬ (x ∈ s2)] then false else true
equal(s1, s2) = if (s1 ⊆ s2 & s2 ⊆ s1) then true else false
size(s) = cardinality(s)

definition
ident_op(f) = ∀x [f(x, x)] &
∀x,y [f(x, y) ⟹ f(y, x)] &
∀x,y,z [f(x, y) & f(y, z) ⟹ f(x, z)] &
∀P ∀x,y [f(x, y) ⟹ (P(x) ≡ P(y))]

end set

In Figure 8, a definition of finite subsets of a type E is given in terms of ordinary set

theory. The notation in the figure is ambiguous, because we wish to use the standard notations for the usual set operations as abbreviations for the operations of the representation algebra as well as for the set operations of the underlying mathematics. The ambiguity is resolved as follows: within the definitions of the operations, the standard set notations refer to the operations of the underlying mathematics, while the **as** clauses in the signature section redefine those notations as abbreviations for the operations of the representation algebra, for external use (i.e., when using representation algebras from the **set** family to define standard models for other data abstractions).

4.3.7 Sequences

We will write **sequence**[E] for the domain of finite sequences of elements of type **E**. An informal definition of sequences in terms of cartesian products is shown in Figure 9. Another definition, using a fixpoint construction, will be sketched in the next section.

Sequences have an exceptional termination condition *bounds*, which is associated with attempts to use elements of the sequence that do not exist. Sequences can be decomposed into the first element and the sequence containing all but the first element, and also into the last element and the sequence of all but the last element, so that neither end of the sequence is preferred with respect to ease of access. Subranges are specified by giving the first and last elements of the subrange in the original sequence. The length of a subrange $s[\, a \,..\, b \,]$ is $1 + b - a$. Subranges with strictly negative lengths are not defined, and an attempt to construct one will result in a *bounds* exception, with no return value.

Figure 9. Sequence

```
type sequence[E]                as Q
requires E : type
with            emptyseq:   → Q                              as ⟨ ⟩
                addfirst:   Q × E → Q                        as arg 2 +| arg 1
                addlast:    Q × E → Q                        as arg 1 |+ arg 2
                butfirst:   Q → Q
                butlast:    Q → Q
                append:     Q × Q → Q                        as arg 1 || arg 2
                subrange:   Q × int × int → Q + ⟨ bounds : ⟩  as arg 1 [ arg 2 .. arg 3 ]
                prefix:     Q × int → Q + ⟨ bounds : ⟩        as arg 1 [ .. arg 2 ]
                suffix:     Q × int → Q + ⟨ bounds : ⟩        as arg 1 [ arg 2 .. ]
                element:    Q × int → E + ⟨ bounds : ⟩        as arg 1 [ arg 2 ]
                first:      Q → E + ⟨ bounds : ⟩
                last:       Q → E + ⟨ bounds : ⟩
                length:     Q → int                          as # arg 1
                empty:      Q → boolean
                equal:      Q × Q → boolean                  as arg 1 = arg 2
representation  Q = ⋃ (i) { i } × E^i                        % i is the length
                    i ≥ 0
restrictions    none
identity        sequence.equal
operations      emptyseq() = ⟨ 0 ⟩
                addfirst(q, e) = ⟨ 1+(#q), e , q[1], ... , q[#q] ⟩
                addlast(q, e) = ⟨ 1+(#q), q[1], ... , q[#q], e ⟩
                butfirst(q) = q[2 .. #q]
                butlast(q) = q[1 .. (#q)-1]
                append(q, r) = if #q = 0 then r
                               else if #r = 0 then q
                               else ⟨ (#q)+(#r), q[1], ... , q[#q], r[1], ... , r[#r] ⟩
                subrange(q, i, j) = if (i < 1) ∨ (j > #q) ∨ (j < i-1) then ⟨ bounds : ⟩
                                    else if j = i-1 then ⟨ 0 ⟩
                                    else ⟨ 1+j-i, q[i], ... , q[j] ⟩
                prefix(q, i) = q[1 .. i]
                suffix(q, i) = q[i .. #q]
                element(q, i) = if (i < 1) ∨ (i > #q) then ⟨ bounds : ⟩
                                else q ↓ (1+i)
                first(q) = q[1]
                last(q) = q[#q]
                length(q) = q ↓ 1
                empty(q) = if #q = 0 then true else false
                equal(q, r) = if #q = #r & ∀i [ 1 ≤ i ≤ #q ⟹ q[i] = r[i] ] then true else false
end sequence
```

4.3.8 Fixpoints

It is convenient at times to introduce algebras whose principal types have a "recursive" structure, such as the algebra of binary trees. While it is possible to define isomorphic images of such algebras using just the machinery introduced so far, by introducing appropriate encodings into the natural numbers, such a strategy does not contribute to the clarity of the resulting specifications. Instead, we will introduce explicit recursive (circular) domain definitions, which are considered as fixpoint equations over the domain of all algebraic structures.

The **representation** component of a specification will always be a domain equation. In cases where the name of the algebra being defined does not appear on both sides of the equation, there is always a unique solution, since we are essentially solving for the fixpoint of a constant transformation. In cases where the representation algebra is defined in terms of itself, there may be many different solutions to the equation. Following Scott[46], we will introduce an ordering, and say that a fixpoint equation denotes the minimal solution with respect to that ordering. We will use the pointwise containment ordering on algebras, denoted by $\sqsubseteq$, and defined below.

Definition 19 Pointwise Containment

Let a and b be algebras. Then $a \sqsubseteq b$ if and only if all of the following conditions hold:

$a.\,\text{typenames} \subseteq b.\,\text{typenames}$,
$\forall \alpha \in a.\,\text{typenames}\ [\ a.\,\text{phyla}_\alpha \subseteq b.\,\text{phyla}_\alpha\ ]$,
$a.\,\text{opnames} \subseteq b.\,\text{opnames}$,
$\forall \beta \in a.\,\text{opnames}\ [\ a.\,\text{operations}_\beta \subseteq b.\,\text{operations}_\beta\ ]$,
$a.\,\text{tcnames} \subseteq b.\,\text{tcnames}$,
$a.\,\text{arglength} \subseteq b.\,\text{arglength}$,
$a.\,\text{argtype} \subseteq b.\,\text{argtype}$,

a.tc $\subseteq$ b.tc,
a.rlength $\subseteq$ b.rlength, and
a.rtype $\subseteq$ b.rtype.
a.pt $\subseteq$ b.pt

If $a \sqsubseteq b$, we will say that a is contained in b. This means that for every phylum of a, b has a phylum of the same name, and for every operation of a, b has an operation with the same name and type. Every phylum of a is a subset of the corresponding phylum of b, and every operation of a is a restriction of the corresponding operation of b. The larger algebra b may have types and operations not present in a. The set of principal types for a must be a subset of the set of principal types of b.

Note that $\sqsubseteq$ is reflexive, transitive, and antisymmetric, and hence is a partial ordering relation. Because $\sqsubseteq$ is antisymmetric, if a minimal solution to a fixpoint equation exists, it must be unique. If we restrict ourselves to expressions built from continuous (with respect to $\sqsubseteq$) transformations on algebras, then the existence of a solution is guaranteed by Kleene's first recursion theorem [23], which also gives us an explicit formula for the solution.

Kleene's first recursion theorem states that if the transformation F is continuous with respect to $\sqsubseteq$, then $F(YF) = YF$, and $YF \sqsubseteq A$ whenever $F(A) = A$, where $YF = \bigsqcup_{i \in \mathbb{N}} F^i(\perp)$, $\bigsqcup$ denotes the least upper bound with respect to $\sqsubseteq$, $F^0(A) = A$, $F^{i+1}(A) = F(F(A))$, and where $\perp$ denotes the least element with respect to $\sqsubseteq$. In other words, if it exists, YF is the least fixpoint of the transformation F. In order to show that YF exists, it is sufficient to show that there is an algebra $\perp$ such that $\perp \sqsubseteq A$ for every algebra A, and that every chain with respect to $\sqsubseteq$ has a least upper bound in the the domain of all exception algebras. It is easy to see that $\perp$ exists: it is the algebra with all components equal to the empty set, containing no phyla and no operations.

Theorem 5 : Every chain with respect to $\sqsubseteq$ has a least upper bound.

Proof : Take pointwise unions, details in Appendix III.
End of Proof

In order to use this result, we need a means of defining continuous transformations. This is also easy, because all of the methods for constructing algebras introduced earlier in this chapter are in fact continuous. The reasoning required to establish continuity is illustrated for the **tuple** transformation.

Theorem 6 : The **tuple** transformation is continuous with respect to $\sqsubseteq$.

Proof : **tuple** preserves pointwise unions for chains of algebras. Details in Appendix III.
End of Proof

Since all constant transformations are continuous, and since the composition of two continuous transformations is continuous, it follows by an easy induction on the depth of the nesting that any expression composed from the constructors for enumerations, tuples, oneofs, sets and sequences defines a continuous transformation. Thus a minimal solution is guaranteed to exist for any domain equation expressible in our specification language.

In order to make sure that the transformations defined earlier in this section are monotonic with respect to $\sqsubseteq$, we have to be a bit more precise about what the transformations are. (If a transformation is continuous with respect to $\sqsubseteq$, then it must also be monotonic with respect to $\sqsubseteq$.)

We will add an implicit parameter to each of the transformations, which specifies the name of the principal type of the algebra resulting from the transformation. The construction of the principal type and of the operations on the principal type has been described above. The subordinate types of each input algebra are included as subordinate types of the output

algebra if and only if they have a distinct name from that given by the implicit parameter. The names of the operations on the principal type are taken from the definitions of the transformations, and prefixed by the name of the principal type to make sure they are distinct from the names of the operations on the subordinate types.

For any composition of **tuple**, **oneof**, **set**, and **sequence** constructions, the implicit name parameters are to be chosen so that every occurrence of each constructor in the expression is given a distinct name parameter, and so that the name parameters are distinct from any of the names of any constant algebras occurring in the expression. With this proviso, any expression that can be formed from the **tuple**, **oneof**, **set**, and **sequence** transformations and any algebra constants will be monotonic with respect to $\sqsubseteq$. It is also easy to see that the new phylum defined by a fixpoint construction will have the same name as its image under the defining transformation, so that the principal type is built up by successive approximations, as usual for a solution to a fixpoint equation. Also note that as defined above, each of our transformations maps complete models into complete models.

An intuitive justification for choosing the minimal solution to a domain equation is that we would like our standard model to be reduced (i.e., free of unnecessary data objects). The explicit solution to the fixpoint equation can also be used to argue that the minimal solution is exactly the solution we would like to obtain, because it contains all of the objects that are finitely constructible using the operations of the representation algebra, and no others. To see this, note that any operation can produce a data object in a domain $F^i(\perp)$ with an index i at most one larger than the index of some domain containing an argument of that operation (or one if there are no arguments). Therefore the results of some finite computation in terms of the primitive operations can produce elements of $F^i(\perp)$ for finite natural numbers i, and all of those

domains are contained in the principal type of YF. Conversely, if our transformation F is such that every element of the principal type of $F(A)$ is finitely constructible whenever all of the elements of the phyla of A are, then so are all of the elements of the phyla of YF, since the principal type of YF is just the union of the principal types of all of the algebras in the chain $F^i(\perp)$.

To illustrate the use of recursively defined representation algebras, consider the definition of immutable binary trees shown in Figure 10. Binary_tree is a family of data abstractions, parameterized by the type of the leaf nodes of the tree. The *leaf* operation creates a leaf containing a given element of type E, where a leaf is a kind of binary_tree. The *tree* operation constructs a composite tree with given left and right subtrees. The *left* and *right* operations return the left and right subtrees of a composite tree, and terminate in the *no_subtree*

Figure 10.

```
type binary_tree[E]                    as T
requires E : type

with            leaf:       E → T
                tree:       T x T → T
                right:      T → T + ⟨ no_subtree : ⟩
                left:       T → T + ⟨ no_subtree : ⟩
                value:      T → E + ⟨ not_leaf : ⟩
                leaf?:      T → boolean

representation  T = oneof[ leaf : E, tuple[ left : T, right : T ] ]

operations      leaf(e) = e in leaf
                tree(x, y) = ⟨ left : x, right : y ⟩ in tree
                right(x) = if is[leaf](x) then ⟨ no_subtree : ⟩ else to[tree](x).right
                left(x) = if is[leaf](x) then ⟨ no_subtree : ⟩ else to[tree](x).left
                value(x) = if is[leaf](x) then to[leaf](x) else ⟨ not_leaf : ⟩
                leaf?(x) = if is[leaf](x) then true else false
end binary_tree
```

exception with no return values if applied to a leaf. The predicate *leaf?* tests a tree to determine whether or not it is a leaf. The *value* operation extracts the element contained in a leaf node of the tree, and it results in a *not_leaf* exception if applied to a composite node.

There is no qualitative difference between defining the operations of a model whose representation algebra is defined by a fixpoint construction and defining the operations of a model whose representation algebra is defined by some finite composition of tuples, oneofs, sets, and sequences. The domain equation specifies the structure of the representation algebra, and implicitly also the operations available on the representation algebra, since each of the transformations mentioned above introduces some operations. For example, since the representation of a binary_tree is a **oneof**, the projections, injections, and domain test predicates of the given oneof type are available for use in defining the operations of binary_tree. This uniformity is a consequence of the fact that the representation algebra is an exact solution to the domain equation.

The fixpoint construction can also be used to construct the natural numbers, and the parameterized family **sequence[E]**. A convenient representation algebra for defining the natural numbers is the solution to the equation

nat = **oneof**[zero : { 0 }, nonzero : nat].

This equation is based on the fact that each natural number is either zero, or it is the successor of some other natural number. Thus zero is represented as the element of the arbitrarily chosen singleton enumeration type { 0 }, and any other natural number is represented by its predecessor injected into the *nonzero* component of the disjoint union. This works because each injection adds a tag to keep the elements of a disjoint union distinct. Thus zero is represented by the

pair ⟨ zero, 0 ⟩, one is represented by the pair ⟨ nonzero, ⟨ zero, 0 ⟩ ⟩, two by the pair ⟨ nonzero, ⟨ nonzero, ⟨ zero, 0 ⟩ ⟩ ⟩, and so on, where the natural number n has n tags equal to *nonzero* and one tag equal to *zero*. A representation algebra suitable for defining **sequence[E]** is defined by the following equation.

seq = oneof[empty : { λ }, nonempty : tuple[first : E, rest : seq]]

The reader is invited to fill in the details of the last two examples, to get some experience in working with recursively defined representation algebras.

Another treatment of recursively defined domains can be found in [26, 25]. We prefer to avoid a category theoretic formulation, on the grounds that the subject can be treated satisfactorily in terms of a more widely known mathematical setting.

4.3.9 System States

In a state machine model, the current system state function is the disjoint union of the current individual state functions for each mutable type. When defining a state machine model in our specification language, we will explicitly construct only the individual state function for the principal type. The individual state functions for the subordinate types are taken from the standard models for the defining abstractions of the subordinate types, and the disjoint unions of the individual state functions required to get a system state function are left implicit.

We provide two abstractions, tokens and states, for use in constructing the principal type and the individual state functions of a state machine model. The interpretation of the principal type of a state machine will always be the principal type of the **token** abstraction, and the set of individual state functions for the principal type of the state machine will always be

state[*D*], where *D* is the set of data states for the state machine.

The **token** and **state**[*D*] abstractions are defined by the standard model shown in Figure II. These abstractions have been defined so that the only property of a token that is externally observable is its identity, by means of the **token$equal** operation. The only way to create a token or to extend the population of the principal type is by means of the state extension operation.

The only way to extract any information from an individual state function is to apply it to a token to get the current data state of that token. If all accesses to the state of a type are limited to the operations provided by the **state** abstraction, then we can be assured that the only state information in a state machine is that associated with the individual data objects, thus enforcing the assumption discussed in Section 3.2 and in Appendix I.

New states can be created by the *init*, *extend*, or *update* operations. The *init* operation creates an empty state. This operation has been included for completeness, since it is required to define the initial state of the state machine. The **state$extend** operation creates a new state in which the data states of all previously existing data objects are unaffected, and in which a new data object has been created, with a given value as its initial data state. This operation is used to describe the dynamic creation of a data object. The **state$update** operation constructs a new state differing from the old one only at a single point in its domain, and it is used to model operations that change the properties of some existing data objects.

In addition, there is an internal function **state$used**, which tests whether a given token has ever been created in a given state. This function may not be used in defining the operations of a standard model, but it is useful in assertions and proofs about dynamic data abstractions (see Section 5.4). Note that the *used* operation will say that an object that has been

Figure 11. Tokens and States

```
module
type token      as T

with            equal:          T x T → boolean

representation  int
restrictions    x such that x ≥ 1
identity        equal

operations      equal(i, j) = if int$equal(i, j) then true else false

end token

type state[D]   as S
requires        D : type

with            init:           → S
                extend:         S x D → S x token
                update:         S x token x D → S + ⟨ undefined_object : ⟩
                apply:          S x token → D                    as arg 1 ( arg 2 )

representation  S = sequence[D]
restrictions    none
identity        sequence$equal

operations      init() = ⟨ ⟩
                extend(s, d) = ⟨ s |+ d, 1 + (#s) ⟩
                update(s, t, d) = if 1 ≤ t ≤ #s then s[ .. (t-1)] |+ d +| s[ (t+1) .. ]
                                                 else ⟨ undefined_object : ⟩
                apply(s, t) = if 1 ≤ t ≤ #s then s(t) else undefined

internal        used:           token x S → boolean
definition      used(t, s) = if 1 ≤ t ≤ #s then true else false

end state
end module
```

---

created and then destroyed (by changing its data state back to **undefined** using **state$update**) has been used, so that in general the *used* operation does not say whether a given object exists

in the current state. (For secure data abstractions, the two notions coincide.)

The reason for defining tokens and states in the same module is to limit access to the operations on tokens. Note that the definitions of the state operations use the representation of tokens, which is available throughout the module, but not outside it. If tokens were defined in a separate module, then the representation would not be accessible, and additional operations on tokens would have to be provided so that the state operations could be defined. However, we do not want modules other than the definition of the state abstraction to have access to any operations on tokens other than *equal*. We freely admit that this is an ad hoc solution, and we refer the reader to [21] for a description of a general access control mechanism for data abstractions.

An individual state function is restricted to take on the special value **undefined** except at a finite number of tokens. This restriction assures us that the domain of system states is countable, even though it is a function space on an infinite domain. One consequence of this is that we have no need of limit constructions or transfinite inductions in reasoning about system states.

A state machine model for the unique_id abstraction is shown in Figure 12. Since the specification has a **data states** component rather than a **representation** component, we know that a state machine is being defined, rather than an exception algebra, and that the representation of the principal type is implicitly defined to be the **token** abstraction defined in Figure 11. In this case the set of data states is a singleton enumeration type. At least one proper data state is needed to distinguish the objects that have been created from those that have not been (and have the data state **undefined**). Unique_ids are immutable (once a unique_id has been created, its properties are fixed forever), so that one proper data state is all that is needed.

Figure 12. Standard Model for Unique-id

```
type unique_id     as U

with               create:    → U
                   equal:     U x U → boolean

data states        D = { null }

operations         create(s)() = extend(s, null)
                   equal(s)(x, y) = ⟨ s, v ⟩
where              v = if tokenequal(x, y) then true else false

end unique_id
```

This example serves to illustrate the state change caused by the creation of a new data object in its purest form. The unique_id abstraction is secure, since there are no operations that destroy unique_ids.

A state machine model for a memory cell containing a single object of type E is shown in Figure 13. Cells are among the simplest mutable data abstractions. The *create* operation returns a new cell with a specified initial contents. Note that the **state**extend operation returns a pair of values, containing the new state and a token representing the newly created object. The new state is the first return value of every operation of a state machine model, and the old state is the first argument. In an implementation, the state is passed around implicitly, while it is explicitly represented in a state machine model. This is reflected in the signature, which has no mention of the state, and describes only the type structure visible externally. The *update* operation returns no data objects, but it produces a new state in which the given cell has a new value for its contents. The *contents* operation returns the current contents of a cell, and the *equal* operation tests to see if two cells are identical. Both of these operations do not modify

Figure 13.

```
type cell[E]        as C
requires            E : type

with                create:     E → C
                    update:     C × E →
                    contents:   C → E
                    equal:      C × C → boolean

data states         D = E

operations          create(s)(e) = extend(s, e)
                    update(s)(c, e) = state[D]update(s, c, e)
                    contents(s)(c) = ⟨ s, s(c) ⟩
                    equal(s)(c1, c2) = ⟨ s, v ⟩
where               v = if token[equal(c1, c2) then true else false

end cell
```

---

the system state. If we view cells as the L-values of the variables of a programming language [cf. 50], then the *equal* operation can be used to determine whether or not there is aliasing between two variables: an assignment to one variable (a cell[update operation) will affect the value of the other if and only if the variables have *equal* L-values.

Note that there is no such thing as an uninitialized cell. If we wanted to define a different cell abstraction, in which cells could be created without being initialized, then we would have to introduce an additional data state to indicate that a cell was uninitialized, since a token with the data state **undefined** represents a cell that has not been created yet. Since we require the operations of a data abstraction to be deterministic, an attempt to find the contents of an uninitialized cell would either have to result in an exception condition, or in some constant default value.

4.4 Well Formed Specifications

A specification is well formed if it denotes some exception algebra or state machine. This will be the case if the requirements described in the following subsections are met. In addition, a reasonably defined data abstraction should satisfy the following two constraints [cf. 41,10].

Every operation of the abstraction d should either take at least one argument from the principal type of d, or it should produce at least one return value (in the normal termination condition) from the principal type of d (or from some principal type if there is more than one). The purpose of this constraint is to rule out functions that have nothing to do with the behavior of the principal type.

There should be at least one operation that produces a value belonging to the principal type which does not take any arguments from the principal type. If this constraint is not met, then there is no way to compute any values of the principal type, and thus the interpretation of the principal type in a reduced model is the empty set.

Note that both of the above constraints can be easily checked given just the signature of the data abstraction. They can be viewed as constraints a structure must satisfy in order to qualify as a meaningful data abstraction.

4.4.1 Type Correctness

All of the expressions in the specification must satisfy the type constraints contained in the signatures of the abstraction being defined, the signature of the representation algebra, and the signatures of the algebras subordinate to either. This means that every operation must be

supplied with the correct number of arguments, and that the definition of each operation must terminate in only those termination conditions specified in the signature, and produce the right number and types of return values for each. This is not a purely syntactic check, because it may require proving that the expression defining an operation terminates in a given termination condition (usually the normal condition).

4.4.2 Representation Consistency

The representation algebra defined by the **representation** section must either be a member of the set of algebras generated by the constructions given earlier in this chapter, or it must have a previously defined standard model. If the representation algebra is defined in terms of a parameterized definition, the constraints specified in the **requires** section of the parameterized definition must be met.

4.4.3 Representation Invariant

If a restriction on the principal type of the representation algebra is specified in the **restrictions** section, then the range of each operation defined must satisfy the restriction. This condition can be established by an inductive argument: assuming that each argument from the principal type satisfies the restriction, show that each return value of each operation satisfies the restriction.

4.4.4 Congruence

If a nontrivial equivalence relation is given in the **identity** section, then it is necessary to show that each operation is consistent with the equivalence relation, in the sense that it maps equivalent arguments into equivalent outputs. This requirement is a necessary condition for the implicit extension of the operations from the representation algebra to the quotient structure to be well defined, as described below.

The operations are explicitly defined as functions that operate on the elements of the principal type of the representation algebra. The model denoted by a specification is in general a quotient structure, and the interpretations of the operations of the data abstraction in that model operate on equivalence classes of elements from the principal type of the representation algebra. The operations can be extended to operate on equivalence classes in the usual fashion. If the operation f takes a single argument from the principal type and returns a single value in the principal type, then the corresponding operation on equivalence classes $f_{\equiv}$ is defined by

$$f_{\equiv}([x]) = [f(x)]$$

where $[x]$ denotes the equivalence class containing the element x. For the relation defined by the above equation to be single valued, and hence a function on equivalence classes, the function f must satisfy the following requirement:

$$x = y \Longrightarrow f(x) \equiv f(y).$$

Note that if $\equiv$ is the same as the logical equality relation on the principal type of the representation algebra, then this requirement is automatically satisfied. An equivalence relation

that satisfies the above constraint is known as a *congruence relation* with respect to f. A definition of $f_{\equiv}$ and of the congruence requirement for a general operation of an exception algebra or a state machine is given below.

Let f be an operation of the abstraction A, $f: a_1 \times \ldots \times a_n \rightarrow \bigsqcup_T R_T$ where $R_T = r_1 \times \ldots \times r_{m(T)}$, and let d be the principal type of A. Let $\equiv$ denote the relation defined by the **identity** section of the specification. Define the equivalence relation eq by

$$eq(x, y) = (x \in d \;\&\; y \in d \;\&\; x \equiv y) \vee (\neg(x \in d) \;\&\; \neg(y \in d) \;\&\; x = y)$$

and define the "equivalence class" ec by

$$ec(x) = \text{if } x \in d \text{ the } [x]_{\equiv} \text{ else } x$$

If $f(x_1, \ldots, x_n) = \langle T, \langle y_1, \ldots, y_{m(T)} \rangle \rangle$, then $f_{\equiv}$ is defined by

$$f_{\equiv}(ec(x_1), \ldots, ec(x_n)) = \langle T, \langle ec(y_1), \ldots, ec(y_{m(T)}) \rangle \rangle$$

and f must satisfy the requirement

$$\forall i : 1 \le i \le n \; [\, eq(x_i, y_i) \,] \Longrightarrow$$
$$tc(f(x_1, \ldots, x_n)) = tc(f(y_1, \ldots, y_n)) \;\&\; \forall j : 1 \le j \le m(T) \; [\, eq(ob_j(f(x_1, \ldots, x_n)) \downarrow j, ob_j(f(y_1, \ldots, y_n)) \downarrow j) \,]$$

4.4.5 Termination

Every operation must be shown to terminate in one of the termination conditions specified in the signature for any set of arguments of the proper type, given that any arguments from the principal type satisfy the restriction given in the **restrictions** section of the specification.

5. Correctness of Implementation

Every well formed implementation of a data abstraction defines an *implementation model* for the data abstraction. The construction of the implementation model is discussed in Section 5.1 below. Our basic definition of correctness is that the implementation model must be behaviorally equivalent to the standard model of the abstraction to be implemented. This definition corresponds to the intuition that there should be no observable difference between the behavior of the implementation and the behavior of the standard model, cast into the framework of deterministic sequential computations.

The classical way to prove the correctness of an implementation with respect to an abstract model specification is to exhibit a homomorphism. In Section 5.2, we show that the classical approach is sound if the standard model and the implementation model are both exception algebras, by showing that the existence of a homomorphism from the implementation model to the standard model implies that the two models are behaviorally equivalent. It was shown in Section 3.3.1 that the classical approach is also complete for the static case, in the following sense: if the standard model is reduced, then there exists a homomorphism from any behaviorally equivalent implementation model to the standard model.

Section 5.3 discusses the case where the standard model is an exception algebra and the implementation model is a state machine. It is shown that a correspondence function analogous to a homomorphism can be used to demonstrate behavioral equivalence.

Section 5.4 discusses the case where the standard model and the implementation model are both state machines. In this case there is no useful analog to the homomorphism theorem of Section 5.2, and proofs of correctness rest directly on the definition of behavioral equivalence.

The proof methodology is illustrated by examples.

5.1 Implementation Models

An implementation of a data abstraction *d* supplies a representation for the principal type, and an algorithm for computing each of the operations. It is relatively easy to construct a model of the abstraction from an implementation, if the representation abstraction and all of the subordinate abstractions have been defined by abstract model specifications.

The principal type of the implementation model is the reachable subset of the principal type of the standard model for the representation abstraction. The reachable subset contains just the elements of the principal type that are computable by some finite (closed) computation in terms of the operations of *d* and the abstractions subordinate to *d*. The interpretation of an operation of the principal type is the function computed by the procedure implementing that operation. The principal type and operations of each abstraction subordinate to *d* are taken from the standard model of the subordinate abstraction.

The implementation model is complete by construction, since it contains interpretations for all subordinate abstractions. The implementation model may or may not be reduced. The construction guarantees that every object of the principal type of the implementation model is reachable. There is no explicit equivalence class structure in the implementation model, so that several distinct implementation objects may represent the same abstract object. The implementation model is reduced if and only if each abstract object has a unique representation in the implementation model.

5.2 Static Specification, Static Implementation

The classical method for demonstrating the correctness of an implementation with respect to a standard model specification is to establish a homomorphism from the implementation model to the standard model. In this section we present a theorem that demonstrates that the classical method is sound for cases where both the standard model and the implementation model are exception algebras.

5.2.1 Homomorphism Theorem

Since an exception algebra has a disjoint union structure not present in the heterogeneous algebras of [I], we have to extend the definition of a homomorphism slightly. A homomorphism between two exception algebras must preserve each operation, which means that the termination conditions of corresponding operation invocations must be the same, and that corresponding return values must be homomorphic images, whenever corresponding arguments are homomorphic images. More precisely, if A and B are two exception algebras with the same signature, then a homomorphism h from A to B is a family of functions $h_\alpha : A.\text{phyla}_\alpha \longrightarrow B.\text{phyla}_\alpha$, where $\alpha \in A.\text{typenames}$, with the following property.

Let $P = A.\text{phyla}$, $F = A.\text{operations}$, $\beta \in A.\text{opnames}$, $n = A.\text{arglength}(\beta)$,
let $a_i = A.\text{argtype}(\beta, i)$ and $x_i \in P_{a_i}$ for each i in the range $1 \le i \le n$,
let $\langle \tau, \langle y_1, \ldots, y_m \rangle \rangle = F_\beta(x_1, \ldots, x_n)$,
where $\tau \in A.\text{tc}(\beta)$, $m = A.\text{rlength}(\tau, \beta)$,
and where $r_j = A.\text{rtype}(\tau, \beta, j)$ and $y_j \in P_{r_j}$ for each j in the range $1 \le j \le m$.
Let $G = B.\text{operations}$.

Then $G_\beta(h_{a_1}(x_1), \ldots, h_{a_n}(x_n)) = \langle \tau, \langle h_{r_1}(y_1), \ldots, h_{r_m}(y_m) \rangle \rangle$.

The "=" in the conclusion refers to the equality relation on abstract objects. For models defined using the specification language introduced in Chapter 4, this relation is given in the identity section of the specification.

Now we can state the homomorphism theorem.

Theorem 7 : Let *M1* and *M2* be complete exception algebra models with a common signature. If there is a homomorphism from *M1* to *M2* which reduces to the identity mapping on the subordinate types, then *M1* and *M2* are behaviorally equivalent.

Proof : By induction on the length of the computation. Details in Appendix III.
End of Proof

The existence of a homomorphism indicates that the interpretation of any closed computation *C* in *M1* is a step by step simulation of the interpretation of *C* in *M2*. Corresponding results (data objects) may have different representations in the two models, but they must have the same properties. Since the homomorphism is required to be the identity mapping on the booleans, the homomorphism property will guarantee that any primitive predicate will give the same truth value for corresponding data objects in *M1* and *M2*.

Note that we are dealing with complete models, which contain the operations of every type subordinate to the principal type in addition to the operations of the principal type. A homomorphism must preserve all of the operations of an exception algebra, including those associated with the subordinate types. It is sufficient to explicitly consider only the operations of the principal type when proving the correctness of a static implementation, because the component of the homomorphism for each of the subordinate types is the identity function, which trivially preserves all of the operations of the defining abstraction of each subordinate type. The requirement that the homomorphism must reduce to the identity mapping on the subordinate types is no restriction in practice, because of the way in which the standard model

and the implementation model are constructed. In both cases the interpretations of the objects and operations of the subordinate types are taken from the standard models of the defining abstractions of the subordinate types. Consequently, the subordinate types have identical interpretations in both models, and the natural correspondence between the two is the identity mapping.

5.3 Static Specification, Dynamic Implementation

In the case where the implementation algebra is a state machine and the standard model is an exception algebra, a correspondence function can be used to establish the behavioral equivalence of the two models in a way entirely analogous to the homomorphisms used in the case where both models are exception algebras. In the rest of this section we present a theorem justifying the use of correspondence functions, and an example to illustrate the procedure for establishing the correctness of a dynamic implementation for a static data abstraction.

The correspondence function that is used to demonstrate the behavioral equivalence of a dynamic model and a static model is not a homomorphism on algebras, even though it must have similar properties. Some of the differences between homomorphisms and correspondence functions are outlined below.

Recall that a homomorphism is a family of mappings, one for each phylum. Each mapping is a function from a phylum of one algebra to the corresponding phylum of the other algebra. The abstract object represented by some implementation object must be completely determined by the identity of the implementation object, since the mapping takes no other arguments. This works well in the static case. In a state machine model, the properties of a

data object will depend not only on the identity of the object, but also on the current system state. Consequently a correspondence function must differ from a homomorphism by taking the system state as an extra argument.

Recall that the principal type of a state machine contains tokens representing all of the data objects that can ever be created. In each system state the population of objects that have been created so far is the subset of the principal type with a proper data state, while the objects that have not been created yet are all mapped into the (improper) data state **undefined** by the system state function. In system states where a given token has the data state **undefined**, the token does not represent any abstract data object, and after an operation is performed that assigns a proper data state to the token, the token represents the newly created data object. To make the correspondence a total function we adopt the following convention. A correspondence function must map a token into the special object **undefined** for any system state for which the token lies outside the current population.

The properties of the newly created object are determined at the time the object is created, and have no particular relation to the identity of the token representing the object. Different computations can lead to states in which a given token has different properties, and in such a case the correspondence function must map the token into different abstract objects in the two states.

The correspondence between the tokens of the implementation model and the abstract objects of the standard model is established by a series of approximations, corresponding to the steps in the computation that create new objects of the principal type. Initially, the population of the implementation model is empty, and the initial correspondence is empty (i.e., in the initial state the correspondence maps every token into the improper object **undefined**). As new

objects are created, the image of the token representing the newly created object changes, from **undefined** in the state just before the object was created, to the abstract object represented by the newly created implementation object in the state just after it was created. For abstractions that do not allow the explicit destruction of data objects, the correspondence functions for the sequence of system states produced by a closed computation are a series of pure extensions. If σ_i represents the state produced by the i-th step of some closed computation, c is a correspondence function, and $i < j$, then it must be the case that

$c(x, \sigma_i) \neq$ **undefined** $\Longrightarrow c(x,\sigma_i) = c(x, \sigma_j)$.

We will refer to this as the *monotonicity property* for correspondence functions. Once an implementation object has been created, and it has come to represent a proper abstract object, the monotonicity property says that the implementation object must continue to represent the same abstract object in all subsequent states. This is just what we would expect; if we create an implementation object and assign it to a variable, we would like to assert that the variable will continue to denote the same (immutable) abstract object as long as we do not assign a new value to the variable. Spontaneous changes in the abstract identity of the value are not acceptable.

A correspondence function must reduce to the identity mapping on the subordinate types, just as for a homomorphism. Note that for the subordinate types the correspondence function is independent of the system state. The abstractions we are considering in this section have static standard models, so that all of the subordinate types must be static, and all of the objects of the subordinate types must therefore exist in all possible system states, in the implementation model as well as in the standard model.

5.3.1 Correspondence Theorem

In this subsection we define correspondence functions precisely, and we present a theorem supporting their usefulness. Let A be a state machine and let B be an exception algebra such that the signature of B is contained in the signature of A. A correspondence function from A to B is a family of functions $c_\alpha : A.\text{phyla}_s \times A.\text{phyla}_\alpha \rightarrow B.\text{phyla}_\alpha$ where $\alpha \in B.\text{typenames}$ and where $s = A.\text{ss}$ is the name of the phylum of system states for the state machine A. A correspondence c must satisfy the following property.

Let $P = A.\text{phyla}$, $F = A.\text{operations}$, $\beta \in B.\text{opnames}$, $n = A.\text{arglength}(\beta)$,
let $a_i = A.\text{argtype}(\beta, i)$ and $x_i \in P_{a_i}$ for each i in the range $1 \le i \le n$,
let $\sigma \in P_s$,
let $\langle \tau, \langle \sigma', y_1, \ldots, y_m \rangle \rangle = F_\beta(\sigma, x_1, \ldots, x_n)$,
where $\tau \in A.\text{tc}(\beta)$, $\sigma' \in P_s$, $m = A.\text{rlength}(\tau, \beta)$,
and where $r_j = A.\text{rtype}(\tau, \beta, j)$ and $y_j \in P_{r_j}$ for each j in the range $1 \le j \le m$.
Let $G = B.\text{operations}$.

Then $G_\beta(c_{a_1}(\sigma, x_1), \ldots, c_{a_n}(\sigma, x_n)) = \langle \tau, \langle c_{r_1}(\sigma', y_1), \ldots, c_{r_m}(\sigma', y_m) \rangle \rangle$,
$x \in P_\alpha \;\&\; \alpha \in A.\text{statenames} \;\&\; x \in \text{population}(\sigma) \Rightarrow c(\sigma, x) = c(\sigma', x)$,
and $x \in P_\alpha \;\&\; \neg\, \alpha \in A.\text{statenames} \Rightarrow c(\sigma, x) = c(\sigma', x)$.

The correspondence property says that the correspondence must preserve all of the operations of the target algebra. Note that the new state σ' produced by the operation of the state machine is used to determine the correspondence between the results of the operation in the state machine and in the exception algebra. A correspondence function must also satisfy the monotonicity requirement, as stated in the last two clauses.

A correspondence function is distinguished from a homomorphism since it takes the system state as an extra argument, and since it satisfies the monotonicity property specified by

the second clause of the conclusion. Since the range of the mapping is an exception algebra, there is no component of the correspondence function for the phylum of system states.

The correspondence theorem assures us that two models are behaviorally equivalent whenever there is a correspondence function from one to the other.

Theorem 8 : Let *M1* be a state machine model and let *M2* be an exception algebra model. If there is a correspondence function from *M1* to *M2* which reduces to the identity mapping on the subordinate types, then *M1* and *M2* are behaviorally equivalent.

Proof : By induction on the length of the computation. Details in Appendix III.
End of Proof

The proof is very similar to the proof of the homomorphism theorem, except that the monotonicity property is required to transfer properties of a data object from the state in which it was created to the state in which it is used as an argument to a subsequent operation.

5.3.2 Simple Example

A very simple example to illustrate a proof of the correctness of a dynamic implementation of a static data abstraction is developed in this subsection. We will consider an implementation of the *intpair* abstraction in terms of arrays of integers. *Intpairs* are immutable pairs of integers, such as might be used to represent rational numbers or gaussian integers. Operations for constructing pairs, and for extracting the left and right components are provided. The *intpair* abstraction is very similar to tuple[right : int, left : int]. An exception algebra model for the *intpair* abstraction is shown in Figure 14.

A state machine model for arrays is shown in Figure 15. Arrays are mutable, and have a variable size. It is not possible to create an array with uninitialized elements. The arrays defined here are a simplified version of CLU arrays, which have more operations.

Figure 14. Pairs of Integers

```
type intpair      as P

with              create:     int x int → P
                  left:       P → int
                  right:      P → int

representation    P = tuple[left: int, right: int]
restrictions      none
identity          tuple$equal

operations        create(x, y) = ⟨ right : x, left : y ⟩
                  left(x) = x.left
                  right(x) = x.right
end intpair
```

An implementation is shown in Figure 16, and the implementation model is shown in Figure 17. The derivation of the implementation model from the implementation is straightforward. The operations of the implementation model are described in the same notation as the operations of the standard model to avoid introducing a host programming language. We claim that it is useful to define the implementation model in this style in doing practical proofs as well, thus separating the issues involved in establishing the correspondence between two different representations for a data abstraction from the problem of proving that a procedure written in a particular programming language implements a particular function.

To prove the correctness of this implementation, we have to exhibit a mapping c and demonstrate that it is indeed a correspondence function. The behavioral equivalence of the standard model and the implementation model will then follow from the correspondence theorem. In order to distinguish the operations of the implementation model from the operations of the standard model in the proof, we will prefix the implementation operations

Figure 15. Arrays

```
type array[E] as A
requires E : type

with        create:   int → A
            addh:     A × int → A
            addl:     A × int → A
            remh:     A → + ⟨ bounds : ⟩
            reml:     A → + ⟨ bounds : ⟩
            store:    A × int × E → + ⟨ bounds : ⟩     as arg 1 [ arg 2 ] := arg 3
            fetch:    A × int → E + ⟨ bounds : ⟩       as arg 1 [ arg 2 ]
            equal:    A × A → boolean                  as arg 1 = arg 2
            low:      A → int
            high:     A → int
            length:   A → int

data states   D = tuple[low: int, e: sequence[E]]
restrictions  none
identity      tuple$equal

operations    create(s)(i) = state[D]$extend(s, ⟨low: i, e: ⟨⟩ ⟩)
              addh(s)(a, x) = ⟨state[D]$update(s, a, ⟨low: s(a).low, e: s(a).e |+ x⟩), a⟩
              addl(s)(a, x) = ⟨state[D]$update(s, a, ⟨low: s(a).low - 1, e: x +| s(a).e⟩), a⟩
              remh(s)(a) = if #(s(a).e) = 0 then ⟨ bounds : s ⟩
                           else ⟨state[D]$update(s, a, ⟨low: s(a).low, e: butlast(s(a).e)⟩), a⟩
              reml(s)(a) = if #(s(a).e) = 0 then ⟨ bounds : s ⟩
                           else ⟨state[D]$update(s, a, ⟨low: s(a).low + 1, e: butfirst(s(a).e)⟩), a⟩
              store(s)(a, i, x) = if s(a).low ≤ i ≤ s(a).low + #(s(a).e) - 1
              then state[D]$update(s, a, ⟨low: s(a).low, e: s(a).e[.. i-1] |+ x +| s(a).e[i+1 ..]⟩)
              else ⟨ bounds : s ⟩
              fetch(s)(a, i) = if s(a).low ≤ i ≤ s(a).low + #(s(a).e) - 1
                               then ⟨ s , s(a).e[i - low + 1] ⟩
                               else ⟨ bounds : s ⟩
              equal(s)(a1, a2) = ⟨s, token$equal(a1, a2)⟩
              low(s)(a) = ⟨s, s(a).low⟩
              high(s)(a) = ⟨s, s(a).low + #(s(a).e) - 1⟩
              length(s)(a) = ⟨s, #(s(a).e)⟩
end array
```

---

with a "↓". To help the reader distinguish elements of the standard model from elements of the implementation model, variables ranging over implementation objects will also be prefixed with

Figure 16. Implementation

representation array[int]

operations create(x, y) = addh(addh(array$create(1), x), y)
left(p) = fetch(p, 1)
right(p) = fetch(p, 2)

Figure 17. Implementation Model

representation array[int]

operations create(s)(x, y) = addh(s2)(p2,y)
where ⟨s2, p2⟩ = addh(s1)(p1, x)
⟨s1, p1⟩ = array$create(s)(1)

left(s)(p) = fetch(s)(p, 1)
right(s)(p) = fetch(s)(p, 2)

---

a "1".

The correspondence function for this example is the following.

$c(s, a) = ⟨left: s(a).e[1], right: s(a).e[2]⟩$

We have shown only the component of the correspondence for the principal type *intpair*. The correspondences for all other types are identity functions.

The proofs for the operations *create* and *left* are shown below. The proof for the operation *right* is similar to the proof for *left*, and is left as an exercise for the reader. The proof relies on the implementation invariant **I** shown below, which is a restriction on the data state of every object representing an *intpair*.

Let x, y be integers,
↓a, ↓p be ↓intpairs,
↓s, ↓s0 be system states for ↓intpairs,

Let $\mathbb{I}$ = ↓s(↓p). low = 1 & *(↓s(↓p). e) = 2

create
Let ⟨↓s, ↓a⟩ = ↓create(↓s0)(x, y).
We have to show that c(↓s, ↓a) = create(x, y).
From the definition of create, create(x, y) = ⟨left: x, right: y⟩.
From the definition of c, c(↓s, ↓a) = ⟨left: ↓s(↓a). e[1], right: ↓s(↓a). e[2]⟩.
Using the definition of tuple$equal, we have to show that
↓s(↓a). e[1] = x and ↓s(↓a). e[2] = y.
From the definition of the array operations create and addh,
↓s(↓a) = ⟨low: 1, e: ⟨x, y⟩⟩ and ↓s(↓p) = ↓s0(↓p) for ↓p ≠ ↓a,
so ↓s(↓a). e[1] = x and ↓s(↓a). e[2] = y.
So c(↓s, ↓a) = create(x, y).
Since ↓a is newly created and ↓s(↓p) = ↓s0(↓p) for all ↓p ≠ ↓a,
the monotonicity property holds.
Since the array operations create and addh can only terminate in the **normal** condition,
c preserves the termination condition of the create operation.
So c preserves the create operation.

Also ↓s(↓a). low = 1 & *(↓s(↓a). e) = 2 and ↓s(↓p) = ↓s0(↓p) for ↓p ≠ ↓a,
so that the implementation invariant holds in state ↓s if it holds in ↓s0.

left
Let ⟨↓s, x⟩ = ↓left(↓s0, ↓a).
Let a = c(↓s, ↓a).
We must show that x = left(a).
By the definition of c, a = ⟨left: ↓s(↓a). e[1], right: ↓s(↓a). e[2]⟩.
By the definition of left, left(a) = ↓s(↓a). e[1].
From the invariant, ↓s0(↓a). low = 1 & *(↓s0(↓a). e) = 2
so ↓s0(↓a). low ≤ 1 ≤ ↓s0(↓a). low + *(↓s0(↓a). e) - 1,
and by the definition of ↓left and array$fetch, ↓s = ↓s0 and
x = ↓s0(↓a). e[1 - 1 + 1] = ↓s0(↓a). e[1].
So x = left(a).
left and ↓left always terminates in the **normal** condition.
So the correspondence c preserves the left operation.
Since ↓s = ↓s0 the monotonicity requirement is trivially satisfied.
The implementation invariant holds since ↓s = ↓s0.

right
Proof left to the reader.

For the purposes of comparison, if immutable sequences had been used as the representation of *intpair* instead of arrays, the homomorphism would have been the following for the analogous representation:

$h(x) = \langle$ left: x[1], right: x[2] $\rangle$.

The proof would have been similar for the immutable case, except that there would have been no need to show the monotonicity property, and no need to argue that the data states of previously existing data objects satisfy the implementation invariant, as we did for the *create* operation. For a mutable implementation, it is important to include this part of the argument, because the implementation invariant is a constraint on the entire system state, rather than just on the images under the new system state of the data objects returned. A correctly implemented operation must preserve the invariant, which means that the invariant must hold with respect to all data objects after the operation is performed. This includes the objects returned by the operation, as well as any others whose state may have changed as a result of the operation.

Note that the proof methodology presented here has no difficulties handling implementations with benevolent side effects. If the correspondence function is many to one, then an operation may change the state of an implementation object without affecting the correctness argument, as long as the image of the implementation object under the correspondence function does not change. Such side effects can be useful in cases where an operation rearranges a data structure to make future operations on that structure more efficient, without changing the externally observable behavior of the structure.

5.4 Dynamic Specification, Dynamic Implementation

The correctness of a dynamic implementation of a dynamic data abstraction can be proved by constructing a *simulation relation*, and by showing that the simulation relation holds for all closed computations. The method of simulation relations is a general solution to the problem of proving the behavioral equivalence of two models, since it can be applied to both static and dynamic models. If the standard model is static, then some simplifications are possible, as illustrated by the homomorphism theorem and the correspondence theorem presented in the previous sections. In this section we consider the fully dynamic case, where the full power of simulation relations is needed.

Recall that each object of a dynamic type is modeled by a token. Tokens have no distinguishing features other than their identities. The properties of a data object represented by a token are modeled by the images of the token under the current system state function. To establish the behavioral equivalence of two models, we must specify the correspondence between the tokens of the two models, and also the relations that must hold between the states of corresponding tokens. The first of the two correspondences is the correspondence relation $\leftrightarrow$ described below, and the second is described by the simulation relation. For a pair of state machine models, the simulation relation is typically defined in terms of the correspondence relation.

Since tokens do not have any distinguishing properties other than their identities, it is generally not possible to describe the correspondence between the tokens of the implementation model and the tokens of the standard model without reference to the computation that produced the current state. The correspondence relation for tokens is easy to describe in terms of the

computation, since the results of corresponding steps of the computation in the two models must correspond to each other. The correspondence relation is defined more precisely as follows.

Definition 20 Correspondence Relation

If the computation C is feasible in models $M1$ and $M2$, if x is the i-th return value of the j-th step of the interpretation of C in $M1$, and if y is the i-th return value of the j-th step of the interpretation of C in $M2$, then we will say that x corresponds to y and we will write $x \leftrightarrow y$.

The correspondence relation applies to system states as well as to data objects. The correspondence relation is syntactic in nature: it is defined in terms of the structure of the computation, without any regard for the meanings of the operations, so that the same definition applies to all data abstractions.

The simulation relation describes the relation that must hold between the states of corresponding data objects in the two models for the objects to have the same externally observable behavior. Examples of simulation relations can be found in the proofs of correctness given in the following sections.

A typical proof of correctness proceeds by induction on the length of the computation, to show that for any closed computation, the termination condition of the last step is the same in both models, and that the simulation relation holds in the final states of the two models. The proof splits up into cases on the type of the last operation of the computation, with one case for each primitive operation.

To establish behavioral equivalence, the simulation relation must imply that corresponding boolean values are equal. Typically the simulation relation will be the conjunction of a number of clauses, where each clause is an implication. The hypothesis of the implication says that a number of pairs of objects have given types and are related by the

correspondence relation $\leftrightarrow$. The conclusion describes the relations that must hold between the identities and states of corresponding objects. The clause stating the standard requirement on boolean values is the following:

$$b \leftrightarrow \downarrow b \Rightarrow b = \downarrow b,$$

where we follow the convention that variables prefixed by a "$\downarrow$" refer to elements of the implementation model, while variables without such a prefix refer to elements of the standard model. Just as we required the homomorphism or correspondence function used in a proof of correctness of a static data abstraction to be the identity mapping on the subordinate types, we will in general require a clause in the simulation relation for each subordinate type, stating that corresponding objects of the subordinate type must be equal.

In order for the induction to go through, the simulation relation must be strong enough to enable the simulation relation to be proved in the final state, given that the simulation relation holds in all previous states. In working out sample proofs, we have found that the definition of the simulation relation usually evolves along with the proof. In the beginning, the simulation relation states just the required constraints on the boolean domain and on the other subordinate types. In considering each operation, it is often found that an additional hypothesis is required to show that the operation preserves the simulation relation defined so far. As clauses are added to the simulation relation, it is of course necessary to go back and show that the other operations preserve the new clause as well. If the implementation is in fact correct, then this process will eventually terminate in a proof that every operation preserves every clause of the simulation relation.

The use of the correspondence function $\leftrightarrow$ is one difference between proofs of

correctness for dynamic abstractions and for static abstractions. Another phenomenon that occurs only for dynamic abstractions is that sometimes it is necessary to consider the operations of the subordinate types in the correctness proof, as well as the operations of the principal type. The operations of any mutable subordinate type must be considered, since they can modify the system state, and since the simulation relation (usually) depends on the system state. The operations of static subordinate types need not be considered, because they cannot change the system state or return objects of the principal type. Since all of the subordinate types of a static abstraction are static, the operations of the subordinate types of any static abstraction need not explicitly enter into the correctness proof.

Any interactions between the observable behavior of a mutable data abstraction and the operations of its mutable subordinate types depend on the mutation of shared data objects. Since the subordinate relation on models is a well founded partial order, it is not possible for any of the operations of a subordinate type to operate directly on any object of the principal type. It is possible for an object x of a subordinate type to share some substructure with an object y of the principal type, so that the externally observable behavior of y can depend on the state of x. Sharing of this kind can occur by construction or by decomposition. In the first case, some primitive operation takes x as an argument and incorporates it into y, where either y is passed as an argument to the operation or created by the operation and returned. In the second case, some primitive operation takes y as an argument and returns the component x.

For an example of a case where an interaction with the operations of a subordinate type is possible, consider the *mset* abstraction described as follows. *Msets* are mutable sets, with the usual set operations, and also an *elements* operation that returns an array containing the elements of the set. In the standard model, the *elements* operation returns a newly created array,

without affecting the state of any *mset*. Consequently, a subsequent assignment to some element of the array returned by the *elements* operation does not affect the contents of the *mset* from which the array was derived. An implementation in which such an assignment did affect the contents of the *mset* would not be behaviorally equivalent to the standard model, but the only way to detect the difference is to perform an array$store operation, which is an operation of a mutable type **subordinate** to *mset*. (Such an incorrect implementation of *mset* is plausible, since it would arise if the programmer chose to represent *msets* as arrays, and in implementing the *elements* operation forgot to return a copy of the array representing the *mset*, rather than the representation itself.) For such an incorrect implementation, it would not be possible to prove that the array$store operation preserves the simulation relation, even though it could be possible to show that every operation of the principal type does preserve the simulation relation.

5.4.1 Simple Example

In this section we present a proof of correctness of an implementation of the *unique_id* abstraction. This is just about the simplest possible data abstraction that requires a state machine model. Recall that *unique_ids* are immutable, but they can be dynamically created. The standard model for the unique_id abstraction is repeated for the reader's convenience in Figure 18. An implementation of *unique_ids* in terms of arrays is shown in Figure 19. In this implementation, we are taking advantage of the fact that array$create always returns a new array (one that has not been used yet in the current computation). The implementation depends only on the identity of the array, so that the contents of the array can be changed arbitrarily without affecting the correctness of the implementation. A newly created array has a length equal to zero, and a specified lower bound for the indices. The standard model for

Figure 18. Standard Model for Unique_id

type unique_id	**as U**	
with	create:	→ U
	equal:	U x U → boolean
data states	D = { null }	
operations	create(s)() = extend(s, null)	
	equal(s)(x, y) = ⟨ s, v ⟩	
where	v = if token$equal(x, y) then true else false	
end unique_id		

Figure 19. Implementation of Unique_id

representation	array[int]
operations	create() = array[int]$create(I)
	equal(x, y) = array[int]$equal(x, y)

---

arrays is shown in Figure 15 in Section 5.3.2. The proof of correctness is shown below. As before, we will prefix operations, objects, and states belonging to the implementation with a "↓" to distinguish them from their counterparts in the standard model.

To prove that unique_id and ↓unique_id are behaviorally equivalent.
Proof by induction on the length of the computation:
Assuming the simulation relation R holds for all computations C such that I ≤ length(C) < N, show that R holds for all C such that length(C) = N.

Let s, s0, sI be system states for unique_id,
↓s, ↓s0, ↓sI be system states for ↓unique_id,
x, xI, y, z be unique_ids
↓x, ↓xI, ↓y, ↓z be ↓unique_ids
b, ↓b be booleans.

Let R ≡ x ↔ ↓x & s ↔ ↓s ⟹ used(x, s) ≡ used(↓x, ↓s)

& x ↔ ↓x & y ↔ ↓y ⟹ (x = y) ≡ (↓x = ↓y)
& b ↔ ↓b ⟹ b = ↓b

Proof by cases on the name of the last operation in C.

Case 1: create

Let s0 ↔ ↓s0.
Let unique_id$create(s0)() = ⟨s1, x1⟩ and ↓unique_id$create(↓s0)() = ⟨↓s1, ↓x1⟩,
so that s1 ↔ ↓s1 and x1 ↔ ↓x1.
By the definition of unique_id$create, state$extend, and state$used,
used(x1, s1) & ¬ used(x1, s0)
and used(z, s0) ≡ used(z, s1) for z ≠ x1.
By the definition of array$create, state$extend, and state$used,
used(↓x1, ↓s1) & ¬ used(↓x1, ↓s0)
and used(↓z, ↓s0) ≡ used(↓z, ↓s1) for ↓z ≠ ↓x1.
So z ↔ ↓z ⟹ used(z, s1) ≡ used(↓z, ↓s1) for any z, ↓z.
So the first clause of R is established for s1, ↓s1.

(lemma 1) if z ≠ x1 and z ↔ ↓z then ↓z ≠ ↓x1:
used(↓x1, ↓s1) & ¬ used(↓x1, ↓s0),
but used(↓z, ↓s1) ≡ used(z, s1) ≡ used(z, s0) ≡ used(↓z, ↓s0),
So ↓z ≠ ↓x1.

(lemma 2) if z = x1 and z ↔ ↓z then ↓z = ↓x1:
Since z = x1, used(z, s1) & ¬ used(z, s0).
By the first clause of R, used(↓z, ↓s1) & ¬ used(↓z, ↓s0).
used(↓z, ↓s0) ≡ used(↓z, ↓s1) for ↓z ≠ ↓x1.
So ↓z = ↓x1.

Let x ↔ ↓x and y ↔ ↓y.

Case 1.1: x ≠ x1, y ≠ x1

By lemma 1, ↓x ≠ ↓x1 and ↓y ≠ ↓y1.
So x ↔ ↓x and y ↔ ↓y in the prefix of the computation C.
So the second clause of R holds by the induction hypothesis.

Case 1.2: x = x1, y ≠ x1

Then x ≠ y.
By lemma 1, ↓y ≠ ↓x1.
By lemma 2, ↓x = ↓x1.
So ↓x ≠ ↓y and the second clause of R holds.

Case 1.3: $x \neq x1$, $y = x1$

Similar to Case 1.2.

Case 1.4: $x = x1$, $y = x1$

Then $x = y$.
By lemma 2, $\downarrow x = \downarrow x1 = \downarrow y$.
So the second clause of R holds.

The third clause of R holds since create and ↓create do not return any boolean values.
So R holds.
Both create and ↓create always terminate in the normal condition.

Case 2: equal

Let $s0 \leftrightarrow \downarrow s0$, $x0 \leftrightarrow \downarrow x0$, and $y0 \leftrightarrow \downarrow y0$.
Let $equal(s0)(x0, y0) = \langle s1, b \rangle$ and $\downarrow equal(\downarrow s0)(\downarrow x0, \downarrow y0) = \langle \downarrow s1, \downarrow b \rangle$.
By the definition of equal, $s1 = s0$ and $b \equiv (x0 = y0)$.
By the definition of ↓equal, $\downarrow s1 = \downarrow s0$ and $\downarrow b \equiv (\downarrow x0 = \downarrow y0)$.
Since R holds in s0, $(x0 = y0) \equiv (\downarrow x0 = \downarrow y0)$, so $b = \downarrow b$, and R holds.
Both equal and ↓equal always terminate in the normal condition.

So ↔ preserves termination conditions and truth values.
Therefore unique_id and ↓unique_id are behaviorally equivalent.

The most important property of a *unique_id* is that it is unique. This is expressed by the second clause of the simulation relation R, which says that two *unique_id*'s have the same representation if and only if the abstract objects they represent are identically the same. The third clause of R is just the standard requirement on boolean values, from which the behavioral equivalence of the two models follows easily. The only operation of *unique_id* that produces a boolean value is *equal*, and for that case the third clause of R follows easily from the second clause and the definition of *equal*. Establishing the second clause is harder, requiring the addition of the first clause to the simulation relation, to strengthen the induction hypothesis. The first clause is based on that fact that corresponding objects in the implementation and in

the standard model are created at the same time, so that either both exist (in states after the abstract object has been created) or both do not exist (in states before the abstract object has been created). Since the object returned by unique_id$create is always newly created (and hence distinct from previously existing objects), and since only one object at a time is created, the unique representation property is preserved.

The proof shown above is a typical example of the argument used to establish a unique representation property, treated in detail. Similar properties will be required in later examples, and we will sketch the proofs without filling in all of the details, assuming that the reader can adapt the argument given in this section.

5.4.2 Typical Example

A simple example of a proof of correctness for a dynamic data abstraction is presented in this section. We have adapted the intset example from [18], without incorporating the bound on the size of a set.[1] A standard model for intsets is shown in Figure 20. Intsets are mutable sets of integers. The *empty* operation creates a new intset, which is initially empty. The *insert* operation inserts a given integer into a given intset, returning no values and changing the state of the intset. The *remove* operation removes a given integer from a given intset. The *has* operation tests to see if a given integer is a member of a given intset.

An implementation of intsets in terms of arrays is shown in Figure 21. This

---

1. If sets with a bounded size are desired, then an exception conditions should be associated with the *insert* operation to indicate when an attempt has been made to exceed the size bound. This will add another case to the proof without further illuminating the methodology, and hence is omitted.

Figure 20. Standard Model for Intset

type intset **as** I

with	empty:	→ I
	insert:	I x int →
	remove:	I x int →
	has:	I x int → boolean
data states	D = set[int]	
restrictions	none	
identity	set$equal	
operations	empty(s)() = extend(s, set$null())	
	insert(s)(x, i) = update(s, set$add(i, s(x)))	
	remove(s)(x, i) = update(s, set$remove(i, s(x)))	
	has(s)(x, i) = ⟨s, set$member(i, s(x))⟩	

end intset

Figure 21. Intset Implementation

representation	intset = array[int]
restrictions	a such that low(a) = l & (low(a) ≤ j , k ≤ high(a) & j ≠ k ⟹ a(j) ≠ a(k))
identity	array$equal
operations	empty() = array[int]$create(l)
	insert(a, i) = if ¬ has(a, i) then addh(a, i)
	remove(a, i) = if has(a, i) then { store(a, find(a, i), a[high(a)] ; remh(a) }
	has(a, i) = if ∃j[low(a) ≤ j ≤ high(a) & a[j]=i] then true else false
definition	find(a, i) = if ∃j[a[j] = i] then j : a[j] = i else 0

---

implementation keeps at most one instance of any given integer in an array, but the order of the elements is arbitrary. The standard model for arrays is shown in Figure I5 in Section 5.3.2. The proof of correctness is shown below. An explanation follows the proof.

To show that intset and ↓intset are behaviorally equivalent.
Proof by induction on the length of the computation:
Assuming R & Π for all computations C such that 1 ≤ length(C) < N,
show R & Π for all C such that length(C) = N.

Let s, s0, s1 be system states for intset
↓s, ↓s0, ↓s1 be system states for ↓intset
x, x1, z be intsets
↓x, ↓x1, ↓z be ↓intsets
i, i1, ↓i, ↓i1, k, n be integers
b, b1, ↓b, ↓b1 be booleans

Let R ≡ ↓s ↔ s & ↓x ↔ x & ↓i ↔ i ⟹ (i ∈ s(x) ≡ ∃j[1 ≤ j ≤ #(↓s(↓x).e) & ↓i = ↓s(↓x).e[j]])
& ↓b ↔ b ⟹ ↓b = b

Let Π ≡ ↓s(↓x).low = 1 & (1 ≤ j , k ≤ #(↓s(↓x).e) & j ≠ k ⟹ ↓s(↓x).e[j] ≠ ↓s(↓x).e[k])

R is the simulation relation and Π is the implementation invariant.

Proof by cases on the name of the last operation of C.

Case 1: create

Let ↓s0 ↔ s0, ↓create(↓s0)() = ⟨↓s1, ↓x1⟩, create(s0)() = ⟨s1, x1⟩
Then we have ↓s1 ↔ s1 and ↓x1 ↔ x1
By the definition of create, s1(z) = s0(z) for z ≠ x1
By the definition of ↓create, ↓s1(↓z) = ↓s0(↓z) for ↓z ≠ ↓x1
So R and Π hold for s = s1, ↓s = ↓s1, x ≠ x1, ↓x ≠ ↓x1
For x = x1 and ↓x = ↓x1 we have
s1(x1) = **set$**null(), so i ∈ s1(x1) is false for all i.
↓s1(↓x1) = ⟨low: 1, e: ⟨⟩⟩, #(↓s(↓x).e) = 0, and 1 ≤ j ≤ 0 is false for all i.
So R holds for the pair of states s1, ↓s1.
↓s1(↓x1).low = 1 and 1 ≤ j , k ≤ 0 is false, so Π holds.
↔ preserves termination conditions since both create and ↓create always terminate in the **normal** condition.

Case 2: insert

Let s0 ↔ ↓s0, x1 ↔ ↓x1, i1 ↔ ↓i1.
Let insert(s0)(x1, i1) = s1, ↓insert(↓s0)(↓x1, ↓i1) = ↓s1.
Then s1 ↔ ↓s1.
We have s0(z) = s1(z) for z ≠ x1, and similarly for ↓s0,
so we have to show R and Π only for x = x1, ↓x = ↓x1, s = s1, ↓s = ↓s1.
By the definition of insert, s1(x1) = s0(x1) ∪ { i1 }.

Case 2.1: il ∈ s0(xl).

Then sl(xl) = s0(xl), and hence sl = s0.
Since R holds in s0, ∃j[1 ≤ j ≤ #(↓s0(↓xl). e) & ↓s0(↓xl). e[j] = ↓il].
So ↓sl = ↓s0 by the definition of ↓insert.
So R and Ⅱ hold by the induction hypothesis.

Case 2.2: ¬ il ∈ s0(xl).

From R in s0, ¬ ∃j[1 ≤ j ≤ #(↓s0(↓xl). e) & ↓s0(↓xl). e[j] = ↓il].
From the definition of ↓insert, ↓sl(↓xl) = ⟨low: l, e: ↓s0(↓xl). e |• ↓i⟩.
∃j[1 ≤ j ≤ #(↓s0(↓xl). e) & ↓s0(↓xl). e[j] = ↓il] ≡
∃j[1 ≤ j ≤ #(↓sl(↓xl). e) - 1 & ↓sl(↓xl). e[j] = ↓il]
and ↓sl(↓xl). e[j] = ↓i for j = #(↓sl(↓xl). e),
so R holds in sl, ↓sl.
From the definition of ↓insert, ↓s(↓xl). low = l.
Ⅱ holds for 1 ≤ j , k ≤ #(↓s0(↓xl). e) = #(↓sl(↓xl). e) - 1 by the induction hypothesis,
and Ⅱ holds for 1 ≤ j < k = #(↓sl(↓xl). e),
since ¬ ∃j[1 ≤ j ≤ #(↓s0(↓xl). e) & ↓s0(↓xl). e[j] = ↓il].
So Ⅱ holds.
↔ preserves termination conditions since both insert and ↓insert
always terminate in the normal condition.

Case 3: remove

Let s0 ↔ ↓s0, xl ↔ ↓xl, il ↔ ↓il.
Let remove(s0)(xl, il) = sl, ↓remove(↓s0)(↓xl, ↓il) = ↓sl.
Then sl ↔ ↓sl.
We have s0(z) = sl(z) for z ≠ xl, and similarly for ↓s0,
so we have to show R and Ⅱ only for x = xl, ↓x = ↓xl, s = sl, ↓s = ↓sl.
By the definition of remove, sl(xl) = s0(xl) - { il }.

Case 3.1: il ∈ s0(xl)

Since R holds in s0, ↓s0 , ∃j[1 ≤ j ≤ #(↓s0(↓xl). e) & ↓s0(↓xl). e[j] = ↓il]
Choose n such that 1 ≤ n ≤ #(↓s0(↓xl). e) & ↓s0(↓xl). e[n] = ↓il.
Ⅱ holds in s0 so n is unique and n = find(↓s0)(↓xl, ↓il).

Case 3.1.1: n = #(↓s0(↓xl). e)

Then from the definition of ↓remove,
↓sl(↓xl) = ⟨low: l, e: ↓s0(↓xl). e[l..n-1]⟩
From Ⅱ with k = n and the previous line,
¬ ∃j[1 ≤ j ≤ #(↓sl(↓xl). e) & ↓sl(↓xl). e[j] = ↓il],
so R holds for i = il.

Since ↓sl(↓xl). e[k] = ↓s0(↓xl). e[k] for 1 ≤ k ≤ n,
for i ≠ il, ∃j[1 ≤ j ≤ #(↓s0(↓xl). e) & ↓s0(↓xl). e[j] = ↓i] ≡
∃j[1 ≤ j ≤ #(↓sl(↓xl). e) & ↓sl(↓xl). e[j] = ↓i].
So R is established in sl, ↓sl.
I in ↓sl follows from I in ↓s0.

Case 3.1.2: n ≠ #(↓s0(↓xl). e)

Then from the definition of ↓remove,
↓sl(↓xl) = ⟨low: l, e: q[1..n-1] |+ q[#q] +| q[n+1..#q-1]⟩
where q = ↓s0(↓xl). e.
From I with k = n and the previous line,
¬ ∃j[1 ≤ j ≤ #(↓sl(↓xl). e) & ↓sl(↓xl). e[j] = ↓il],
so R holds for i = il.
Since ↓sl(↓xl). e[k] = ↓s0(↓xl). [k] for 1 ≤ k ≤ n - 1 and n + 1 ≤ k ≤ #q - 1,
and ↓sl(↓xl). e[n] = ↓s0(↓xl). e[#q],
for i ≠ il, ∃j[1 ≤ j ≤ #(↓s0(↓xl). e) & ↓s0(↓xl). e[j] = ↓i] ≡
∃j[1 ≤ j ≤ #(↓sl(↓xl). e) & ↓sl(↓xl). e[j] = ↓i].
So R is established in sl, ↓sl.
I in ↓sl follows from I in ↓s0.

Case 3.2: ¬ il ∈ s0(xl)

Then sl(xl) = s0(xl) - { il } = s0(xl) so sl = s0.
Since R holds in s0, ¬ ∃j[1 ≤ j ≤ #(↓s0(↓xl). e) & ↓s0(↓xl). e[j] = ↓il],
so ↓sl = ↓s0, by the definition of ↓remove and ↓has.
So R and I hold in sl, ↓sl.

↔ preserves termination conditions since both remove and ↓remove always terminate in the normal condition.

Case 4: has

Let s0 ↔ ↓s0, xl ↔ ↓xl, il ↔ ↓il.
Let has(s0)(xl, il) = ⟨sl, bl⟩, ↓has(↓s0)(↓xl, ↓il) = ⟨↓sl, ↓bl⟩.
Then sl ↔ ↓sl and bl ↔ ↓bl.
From the definitions of has and ↓has, sl = s0 and ↓sl = ↓s0.
So I holds.
We need to show that bl = ↓bl.
By the definition of has, bl = il ∈ sl(xl).
By the definition of ↓has, ↓bl = ∃j[1 ≤ j ≤ #(↓s0(↓xl). e) & ↓s0(↓xl). e[j] = ↓i].
By the first clause of R, bl = ↓bl.
↔ preserves termination conditions since both has and ↓has always terminate in the normal condition.

Since ↔ preserves termination conditions and the simulation relation,

all computations are equifeasible in intset and in ⊥intset, and each computation producing a boolean value produces the same value in both models. So intset and ⊥intset are behaviorally equivalent.

The only primitive *intset* operation that can produce a boolean value is *has*, and the relationship required for the *has* operation to give the same results in both models is expressed by the first clause of the simulation relation R. The implementation invariant I expresses a restriction on the implementation structures that must be maintained by the operations of the implementation. Note that the implementation invariant does not mention any objects of the standard model, in contrast to the simulation relation, which is concerned with the relations between the two models. The implementation invariant says that all of the elements of the array representing an *intset* must be distinct, and that the low bound of the array must be always equal to 1 (recall that arrays can grow and shrink from both ends). The implementation invariant is needed in the proof to show that the *remove* operation preserves the simulation relation.

Whenever there is a state transition caused by the invocation of an operation of the state machine, we have to reestablish that the properties required for our proof of correctness are still true in the final state. There can be no simple general rule for transferring properties from one state to the next, because there is no simple syntactic relation between the text specifying an operation and the set of data objects that can be affected by the operation. In general, the effects of an operation are not limited to the data objects that are passed as arguments to the operation, because the data state of an object can contain other data objects, which in turn can have data states containing still more data objects. An invocation of an operation can potentially affect every object in the reachability closure of the arguments, which

can vary from one state to the next. Consequently we must establish the invariance of properties of data objects with respect to state transitions on a case by case basis.

As can be seen in the proof above, explicitly arguing that each property is transferred from one state to the next need not lead to unmanageable complexity. In a correctness proof we are typically trying to show that the simulation relation and the implementation invariant remain true in spite of any state transitions that may be caused by the operations of the data abstraction we are trying to verify. In the example above, the arguments are very simple, since there is no potential for data sharing between *intsets*. In the example shown in the next section, there is potential sharing among the objects of the principal type, so that the arguments required to show that the simulation relation is preserved by a state transition have more content.

5.4.3 Sophisticated Example

A sophisticated example, consisting of the nonstandard implementation for mutable lists discussed at the end of Chapter 3, is presented in this section. This example treats a mutable abstraction whose objects may share subcomponents. The implementation is not reduced, so that more than one object (token) in the implementation may represent the same abstract mutable list. The standard model for mutable lists is shown in Figure 22. The implementation model for mutable lists is shown in Figure 23. We have defined the implementation model in the same notation as the standard model in order to keep the example as simple as possible. Strictly speaking, this example shows a proof of the behavioral equivalence of two models. The proof of correctness is outlined below. The proof for the *cdr* operation is very similar to the proof of the *car* operation, and similarly for *rplaca* and *rplacd*,

Figure 22. Standard Model for Mutable Lists

```
type list as L

with        nil:        → L
            cons:       L × L → L
            car:        L → L + ⟨ no_car : ⟩
            cdr:        L → L + ⟨ no_cdr : ⟩
            rplaca      L × L → L + ⟨ no_car : ⟩
            rplacd:     L × L → L + ⟨ no_cdr : ⟩
            eq:         L × L → boolean

data states D = oneof[null: { nil }, pair: tuple[l: L, r: L]]
restrictions none
identity    token$equal

operations  nil(s)() = state[D]$extend(s, nil in null)
            cons(s)(x, y) = state[D]$extend(s, ⟨l: x, r: y⟩ in pair)
            car(s)(x) = if is[pair](s(x)) then ⟨s, to[pair](s(x)).l⟩
                                          else ⟨no_car : s⟩
            cdr(s)(x) = if is[pair](s(x)) then ⟨s, to[pair](s(x)).r⟩
                                          else ⟨no_cdr : s⟩
            rplaca(s)(x, y) = if is[pair](s(x))
                              then ⟨state[D]$update(s, x, ⟨l: y, r: to[pair](s(x)).r⟩ in pair), x⟩
                              else ⟨no_car: s⟩
            rplacd(s)(x, y) = if is[pair](s(x))
                              then ⟨state[D]$update(s, x, ⟨l: to[pair](s(x)).l, r: y⟩ in pair), x⟩
                              else ⟨no_cdr: s⟩
            eq(s)(x, y) = token$equal(x, y)
end list
```

---

so that only one proof is given for each pair of operations, and the other is left to the reader.

An explanation is given after the text of the proof.

To show that list and ↓list are behaviorally equivalent.
Proof by induction on the length of the computation:
Assuming R holds for all computations C such that $1 \leq \text{length}(C) < N$,
show that R holds for all C such that length(C) = N.

Let s, s0, s1 be system states for list,
↓s, ↓s0, ↓s1 be system states for ↓list,

Figure 23. Implementation Model for Mutable Lists

```
data states D = cell[oneof[null: { nil }, pair: tuple[l: L, r: L]]]

operations    nil(s)() = state[list]!extend(state[cell]!extend(s, nil) in null))
              cons(s)(x, y) = state[list]!extend(state[cell]!extend(s, ⟨l: x, r: y⟩ in pair))
              car(s)(x) = if is[pair](s(s(x)))
                          then ⟨s, to[pair](s(s(x))).l⟩
                          else ⟨no_car: s⟩
              cdr(s)(x) = if is[pair](s(s(x)))
                          then ⟨s, to[pair](s(s(x))).r⟩
                          else ⟨no_cdr: s⟩
              rplaca(s)(x, y) = if is[pair](s(s(x))
                                then state[list]!extend(
                                     state[cell]!update(
                                      s, s(x), ⟨l: y, r: to[pair](s(s(x))).r⟩ in pair ),
                                     s(x) )
                                else ⟨no_car: s⟩
              rplacd(s)(x, y) = if is[pair](s(s(x))
                                then state[list]!extend(
                                     state[cell]!update(
                                      s, s(x), ⟨l: to[pair](s(s(x))).l, r: y⟩ in pair ),
                                     s(x) )
                                else ⟨no_cdr: s⟩
              eq(s)(x, y) = token!equal(s(x), s(y))
```

---

w, x, y, z, x0, y0 be lists
↓w, ↓x, ↓y, ↓z, ↓x0, ↓y0 be ↓lists,
b, ↓b be booleans,
↓c be a cell.

Let R ≡ (x ↔ ↓x & s ↔ ↓s & is[null](s(x)) ⟹ is[null](↓s(↓s(↓x)))
& (x ↔ ↓x & s ↔ ↓s & is[pair](s(x)) ⟹ is[pair](↓s(↓s(↓x)))
& s(x).l ↔ ↓s(↓s(↓x)).l
& s(x).r ↔ ↓s(↓s(↓x)).r)
& (x ↔ ↓x & y ↔ ↓y & s ↔ ↓s ⟹ (x = y) ≡ (↓s(↓x) = ↓s(↓y)))
& (b ↔ ↓b ⟹ b = ↓b)

Proof by cases on the name of the last operation of C.

Case l: nil

Let s0 ↔ ↓s0.

Let nil(s0)() = ⟨sl, w⟩ and ↓nil(↓s0)() = ⟨↓sl, ↓w⟩.
Then sl ↔ ↓sl and w ↔ ↓w.
sl(z) = s0(z) for z ≠ w, and similarly for ↓sl.
So we need to show R only for x = w, ↓x = ↓w.
By the definition of nil, is[null](sl(w)).
By the definition of ↓nil, is[null](↓sl(↓sl(w))).
So the first clause of R holds.
The second clause is trivially true for x = w, ↓x = ↓w
since the hypothesis of the implication is false.
Since w and ↓sl(↓w) are newly created, the third clause of R holds.
Both nil and ↓nil always terminate in the normal condition.

Case 2: cons

Let s0 ↔ ↓s0, x0 ↔ ↓x0, y0 ↔ ↓y0.
Let cons(s0)(x0, y0) = ⟨sl, w⟩ and ↓cons(↓s0)(↓x0, ↓y0) = ⟨↓sl, ↓w⟩.
Then sl ↔ ↓sl and w ↔ ↓w.
sl(z) = s0(z) for z ≠ w, and similarly for ↓sl.
So we need to show R only for x = w, ↓x = ↓w.
By the definition of cons, is[pair](sl(w)), sl(w).l = x0, and sl(w).r = y0.
By the definition of ↓cons, is[pair](↓sl(↓sl(↓w)), ↓sl(↓sl(↓w)).l = ↓x0, and ↓sl(↓sl(↓w)).r = ↓y0.
The first clause of R is trivially true.
Since x0 ↔ ↓x0 and y0 ↔ ↓y0, the second clause of R holds.
Since w and ↓sl(↓w) are newly created, the third clause of R holds.
Both cons and ↓cons always terminate in the normal condition.

Case 3: car

Let s0 ↔ ↓s0 and x0 ↔ ↓x0.

Case 3.1: is[pair](s0(x0))

Let car(s0)(x0) = ⟨sl, w⟩ and ↓car(↓s0)(↓x0) = ⟨↓sl, ↓w⟩.
Then sl ↔ ↓sl and w ↔ ↓w.
By the definition of car, sl = s0 and w = s0(x0).l.
By the definition of ↓car, ↓sl = ↓s0 and ↓w = ↓s0(↓s0(↓x0)).l
Since sl = s0 and ↓sl = ↓s0, R holds in sl, ↓sl for x, y ≠ w.
Since R holds in s0, ↓s0, is[pair](↓s0(↓s0(↓x0))) and s0(x0).l ↔ ↓s0(↓s0(↓x0)).l.
So w ↔ ↓w for the prefix of C.
So R holds for sl, ↓sl.
Both car and ↓car terminate in the normal condition for this case.

Case 3.2: is[null](s0(x0))

Then since R holds in s0, is[null](↓s0(↓s0(↓x0))).

Let car(s0)(x0) = s1 and ↓car(↓s0)(↓x0) = ↓s1.
Then s1 ↔ ↓s1.
By the definition of car and ↓car, s1 = s0 and ↓s1 = ↓s0, so R holds.
Both car and ↓car terminate in the no_car condition for this case.

Case 4: cdr

Similar to Case 3, proof left to the reader.

Case 5: rplaca

Let s0 ↔ ↓s0, x0 ↔ ↓x0, and y0 ↔ ↓y0.

Case 5.1: is[pair](s0(x0))

From R, is[pair](↓s0(↓s0(↓x0))).
Let ⟨s1, w⟩ = rplaca(s0)(x0, y0).
Let ⟨↓s1, ↓w⟩ = rplaca(↓s0)(↓x0, ↓y0).
Then s1 ↔ ↓s1 and w ↔ ↓w.
By the definition of rplaca,
s1(z) = s0(z) for z ≠ x0 = w.
By the definition of ↓rplaca,
↓s1(↓z) = ↓s0(↓z) for ↓z ≠ ↓w and ↓s1(↓c) = ↓s0(↓c) for ↓c ≠ ↓s0(↓w) = ↓s0(↓x0).
R holds in s0, ↓s0, and from the third clause of R,
z ↔ ↓z & z ≠ x0 ⟹ ↓s0(↓z) ≠ ↓s0(↓x0).
So ↓s1(↓s1(↓z)) = ↓s0(↓s0(↓z)) for ↓z ↔ z ≠ x0.
So R holds for x ≠ x0.
The first clause of R holds for x = x0 = w since ¬ is[null](s1(w)).
From the definition of rplaca, w = x0.
From the definition of ↓rplaca, ↓s1(↓w) = ↓s0(↓x0),
and ↓s1(↓z) = ↓s0(↓z) for ↓z ≠ ↓w.
So the third clause of R in s1, ↓s1 follows from R in s0, ↓s0.
From the definition of rplaca, s1(w) = ⟨l: y0, r: s0(x0)• r⟩.
Suppose x = x0 = w.
Then from the third clause of R,
x ↔ ↓x ⟹ ↓s1(↓x) = ↓s1(↓w), and ↓s1(↓s1(↓x)) = ↓s1(↓s1(↓w)).
From the definition of ↓rplaca, is[pair](↓s1(↓s1(↓w))) and
↓s1(↓s1(↓w)) = ⟨l: ↓y0, r: ↓s0(↓s0(↓x0))• r⟩.
We have y0 ↔ ↓y0, and since R holds in s0, ↓s0,
s0(x0)• r ↔ ↓s0(↓s0(↓x0))• r.
So the second clause of R holds in s1, ↓s1.
So R holds.
Both rplaca and ↓rplaca terminate in the normal condition for this case.

Case 5.2: is[null](s0(x0))

Let rplaca(s0)(x0, y0) = s1 and ↓rplaca(↓s0)(↓x0, ↓y0) = ↓s1.
By the definitions of rplaca and ↓rplaca, s0 = s1 and ↓s0 = ↓s1, so R holds.
Both rplaca and ↓rplaca terminate in the no_car condition for this case.

Case 6: rplacd

Similar to Case 5, proof left to reader.

Case 7: eq

Let s0 ↔↓s0, x0 ↔ ↓x0, and y0 ↔ ↓y0.
Let eq(s0)(x0, y0) = ⟨s1, b⟩ and ↓eq(↓s0)(↓x0, ↓y0) = ⟨↓s1, ↓b⟩.
By the definition of eq, s1 = s0 and b ≡ (x0 = y0).
By the definition of ↓eq, ↓s1 = ↓s0 and ↓b ≡ (↓s0(x0) = ↓s0(y0)).
Since R holds in s0, x0 ↔ ↓x0, and y0 ↔ ↓y0, (x0 = y0) ≡ (↓s0(↓x0) = ↓s0(↓y0)).
So b = ↓b, and R holds.
Both eq and ↓eq terminate in the normal condition.

So list and ↓list are behaviorally equivalent.

The mutable list example was chosen to illustrate several issues arising from the sharing of mutable data objects. Since we have made a strict distinction between the identity of a token and its state, there is no notational difficulty in stating that one object is a subcomponent of the states of several other objects (i.e., that the first object is shared by the latter objects). Note the use of the correspondence relation ↔ in the conclusion of the second clause of the simulation relation R, to indicate that the identities of the components of a non-null list must correspond in the two models.

The example illustrates a case where there may be many distinct representations for the same mutable object. Every time a *rplaca* or *rplacd* operation is performed on a list, a new representation object for that list is created in the implementation. Despite the multiple representations, the externally observable behavior of mutable lists is correctly realized in the implementation, so that the non-uniqueness of the representation used by the implementation is

not externally observable. Whenever the state of a list is modified by a *rplaca* or *rplacd* operation, the change is reflected in the states of *all* of the representations for the abstract list that was modified, and not just in the particular representation that is returned as the result. This is accomplished by introducing an extra level of indirection: the state of a list in the implementation model is a cell containing the abstract state of the list. In our notation, if $s \leftrightarrow \downarrow s$ are two corresponding states and if $x \leftrightarrow \downarrow x$ are two corresponding lists, then the abstract state $s(x)$ corresponds to the concrete state $\downarrow s(\downarrow s(\downarrow x))$, where $\downarrow s(\downarrow x)$ denotes the identity of the shared cell. The cell is shared by all of the representations of the same abstract list, and all of the relevant state information is contained in this cell, so that any state changes are automatically reflected in all "copies" of the list object. The *eq* operation computes the identity relation on abstract lists, rather than the identity relation of the implementation model, which is not externally observable. The identity relation on abstract lists is described by the third clause of the simulation relation R. Note that the implementation depends critically on the fact that the data state of a token representing a list (the identity of the shared cell) never changes, although the data state of the data state of the token (the contents of the cell) may change. It is easy to check that this property is maintained, since none of the $\downarrow$list operations applies a state-update operation directly to a token representing a list.

The interesting part of the proof is case 5.1, where the normal termination of the state changing *rplaca* operation is treated. Note the use of the third clause of the simulation relation R to implicitly describe the set of representation objects affected by the operation. Implementation objects other than those passed as arguments can be affected by a *rplaca* operation, due to the shared mutable cell in the state of a list in the implementation model. In the argument to establish that the *rplaca* operation does not damage the simulation relation for

objects other than those passed as arguments, the relation given by the third clause of R is used to distinguish between the set of objects that is supposed to be affected by the operation from those objects that are not supposed to be affected. It is just as important to establish that all of the objects whose states are supposed to be affected by the operation reflect the change as it is to show that the objects that are not supposed to be affected retain their previous properties.

While it may be difficult to derive a description of the set of objects that is supposed to be affected by a given operation from an implementation of an arbitrary mutable data abstraction, it is important to make this set explicit, because errors stemming from hidden interactions due to unintended sharing relations are very difficult to track down. The designer should therefore pay explicit and careful attention to the characterization of the set of data objects that should be affected by an operation during the design of the implementation. The intended restrictions on the sharing relationships should be written down as part of the design process, for later reference and for possible use in proofs. This suggestion is analogous to the suggested practice of developing loops together with the associated loop invariants. The suggestion is motivated by the fact that the required information must be informally considered by the designer anyway, and that it is easier to formalize a familiar but informal notion than it is to derive the required properties from an unfamiliar implementation.

6. Conclusions

In this chapter we review the concepts central to this work, present a comparison of the algebraic and abstract model specifications, and suggest some directions for future research.

6.1 Central Concepts

We have been concerned with treating potentially shared mutable data. This orientation has lead us to adopt an object oriented viewpoint, and to define the correctness of an implementation of a data abstraction in terms of the behavioral equivalence of the implementation and the standard model. To prove the correctness of an implementation, we have found it necessary to replace the representation function introduced by Hoare [18] with the simulation relation. We have also found that a form of computation induction is an appropriate method for proving properties of mutable data abstractions.

6.1.1 Data Objects

In this work we have adopted an object oriented viewpoint, rather than the more conventional variable oriented view. This choice was motivated by our desire to treat shared mutable data. If there is no sharing of data, then a change in the state of a data object can affect at most one variable, and the change can be modeled as the assignment of a new immutable value to the affected variable. If data can be shared, then a change in the state of a mutable data object can affect arbitrarily many variables, so that the simplicity of the variable oriented viewpoint breaks down.

In our approach, states are associated with data objects as well as with variables. A

mutable data object is modeled as a featureless token, which serves to identify the object. The value of a variable is a data object (or token). The value of a variable is affected by assignment statements. The system state function maps each token into its current data state. For most mutable data abstractions all of the interesting properties of a mutable data object other than its identity are subject to change, and are represented by the data state associated with the object by the current system state function. The state of a data object of a given type can be affected by the primitive operations of the type.

By introducing an extra level of indirection in our model, we achieve localized descriptions of operations that modify potentially shared data. If two variables share the same data object, then they denote the same token, and any change in the data state of that token will be reflected in both variables. After such a state transition, both variables retain their original values, since the identity of the shared data object is not changed, but the properties of the shared object are different in the new state.

6.1.2 Behavioral Equivalence

The concept of behavioral equivalence of models is central to this work. Two models are behaviorally equivalent if every computation results in the same termination condition in both models, and if any computation with a boolean result yields the same value in both models. This formal characterization of the externally observable behavior of a model is intuitively satisfying, since it says that two models are behaviorally equivalent if they have the same externally observable properties. The characterization is also useful because it allows us to compare models with quite different internal structures. We have to examine only the names of termination conditions and boolean values to apply our definition of behavioral equivalence.

The representation of the objects of all other types is not explicitly mentioned, and can be different in the two models to be compared. System state functions are never explicitly compared, and it does not matter whether a model has a phylum of system states or not. It is quite possible for a state machine model (with system states) to be behaviorally equivalent to an exception algebra model (without system states).

An implementation of a data abstraction is correct if and only if it is behaviorally equivalent to the standard model of the abstraction. We feel that this definition of correctness with respect to an abstract model specification is the right one to use, because it reduces to the classical one (existence of a homomorphism) for the case where both the standard model and the implementation model are static (see theorems 4 and 7), and because it applies also to dynamic models, whereas the classical criterion does not.

6.1.3 Simulation Relations

We have developed a method based on simulation relations for proving the behavioral equivalence of two models. The method can be used to prove the correctness of an implementation of a data abstraction with respect to an abstract model specification. The method is applicable to all models satisfying the assumptions set down at the beginning of this work, but it is most useful in the case where both models are dynamic. Simpler methods based on correspondence functions and homomorphisms are available for the cases where one or both models are static, as described in Chapter 5. Simulation relations and correspondence functions were introduced because it was found that homomorphisms do not suffice for dynamic models.

A simulation relation describes the relation that must hold between the representations and data states of corresponding objects in the implementation and standard models in order

for the externally observable behavior of the objects to be the same. To show that two models are behaviorally equivalent, a simulation relation is explicitly constructed, and it is established that the simulation relation holds for all reachable states by induction over all computation sequences. To establish behavioral equivalence, the simulation relation must imply that corresponding operations on corresponding objects result in the same termination conditions and boolean values. The simulation relation must also be strong enough to establish all of the properties of the inputs that the operations depend on, so that the induction will go through.

Simulation relations are defined in terms of the correspondence relation $\leftrightarrow$, which relates the identities of corresponding data objects in the two models. $\leftrightarrow$ is defined in terms of the computation sequence, by saying that the results of corresponding steps of the computation in the two models are related by $\leftrightarrow$. Since the tokens of a dynamic model are anonymous, and since operations that create new data objects result in tokens unrelated to previously known tokens, the only generally applicable method for establishing the correspondence is to appeal to the history of the computation. A simulation relation has the same purpose as a homomorphism, but it cannot be defined as a function in the dynamic case because of the dependence on the history of the computation. In the static case, a simulation relation would require that objects related by $\leftrightarrow$ are homomorphic images, but since there is no need to separate the identity of an object from its properties in the static case, the homomorphism can be used in the proof directly, without introducing the $\leftrightarrow$ relation.

6.1.4 Proving Theorems about Data Abstractions

In the main body of this work we have concentrated on proving the correctness of an implementation of a data abstraction with respect to a standard model specification. This is only half of the process required to verify programs that use data abstractions. The other half of the process involves proving that the invocations of the operations of a data abstraction in a program written using the abstraction have the specified effect.

The intended behavior of a program is typically described by giving assertions expressing the relations that must hold between the data objects manipulated by the program at various points in its execution. For programs that use data abstractions, the assertions will be written in terms of the primitive operations of the abstraction. For dynamic abstractions, the system state must be explicitly included in the assertions, so that the operations can be treated as functions, and used without regard for the context in which they appear (i.e., there are no side effects in the assertion language).

The problem of showing that a program satisfies its assertions can be reduced to the problem of proving theorems about the data abstractions it uses, by using an axiomatic definition of the control constructs of the programming language to eliminate the program texts from the correctness requirements. The theorems derived from the annotated program texts, which must be proved in order to establish its correctness, are called verification conditions. The process of deriving the verification conditions from an annotated program text has been extensively treated in the literature on program verification for the case where the data abstractions used by the program are well understood domains such as the integers. The process is not significantly affected by the introduction of static user defined data abstractions.

The introduction of dynamic abstractions raises the problem of introducing symbolic names for the intermediate system states, which are implicit in the program text but which are required in the assertions. This process will require a flow analysis of the program. Previous work on automatic verification of programs operating on mutable data [51, 32] has not explicitly introduced states into assertions, avoiding this issue. While we have not investigated the problem in detail, we foresee no essential difficulty in producing verification conditions for programs that use mutable data abstractions.

The problem of proving the verification conditions based on an abstract model specification presents no methodological problems, although just about any interesting data domain has theorems which are hard to prove. It is sufficient to prove the interpretations of the verification conditions in the standard models of the data abstractions used by the program, since behavioral equivalence guarantees that all of the ground terms composed from the primitive operations of the abstraction will have the same truth values in both the standard model and any behaviorally equivalent implementation model. Since quantifiers can be restricted to range over only the computable objects of a data abstraction, behavioral equivalence implies that any assertion will have the same truth values in both models.

6.1.5 Computation Induction

In doing the proofs of correctness of implementation in Chapter 5, we have used a form of computation induction to establish that the simulation relation holds for all (reachable) objects and states of an abstraction. This technique is useful for proving properties of dynamic data abstractions, and performs the same function as the generator induction rule for static data abstractions. There are two essential differences between the two kinds of induction.

The generator induction rule requires us to show that all of the operations of a data abstraction preserve the property to be proved. To prove a property of the data abstraction *d* using the computation induction rule, we must show that the operations of any mutable abstraction subordinate to *d* preserve the property, in addition to the operations of *d*. This is necessary because the operations of the mutable subordinate types can cause state transitions that can affect the truth of an assertion involving objects of type *d* (see Chapter 5).

The generator induction rule requires us to show that the objects returned by each operation satisfy the property we are trying to prove. The computation induction rule also requires us to show that all of the values resulting from each operation satisfy the property we are trying to prove, including the new system state function that is an implicit result of each operation of a state machine. Since the system state function describes the current states of all of the data objects in the system, we have to show that the property we are trying to prove holds in the new system state for all data objects, and not just for the data objects that were passed as arguments to the operation or that were returned as results. This is necessary because an operation can cause state changes in objects that were not passed as arguments, but which are reachable from the arguments.

6.2 Algebraic vs. Abstract Model Specifications

In this section we point out some of the relations between the abstract model and the algebraic specification techniques, and present a critical comparison between the two techniques.

6.2.1 Relation of the Techniques

The algebraic and the abstract model techniques are both concerned with specifying the behavior of a data abstraction, and hence both are dealing with the same class of mathematical structures, although there are slight technical differences in the way in which different researchers define the class. There are several well known results relating an algebraic specification to the class of models satisfying the specification.

One of the main algebraic results relevant to the algebraic specification technique [9] is a uniform construction of a canonical model for any axiomatization consisting of a set of equations, where the expressions on both sides of each equation are composed from the operations of the data abstraction. The model resulting from this construction is a quotient structure, whose elements are equivalence classes of expressions, where two expressions are equivalent if one is derivable from the other from the axioms in finitely many steps. This theorem establishes a connection between the proof theory of an algebraic specification and an algebraic model for the specified abstraction. The theorem allows us to view an algebraic specification as a prescription for constructing a standard model for the data abstraction that is specified, so that an algebraic specification can be considered either as an axiomatization or as the definition of a standard model.

Another important algebraic result is that the canonical model constructed as described above is an initial algebra in the category of algebras satisfying the axioms [9], which means that there is a homomorphism from the initial algebra to any other algebra in the category. In view of theorem 7, and the existence of the homomorphisms guaranteed by the initiality property, all of the elements of the category are behaviorally equivalent to the initial

algebra. In view of theorem 4, if the initial algebra is reduced, then there is a homomorphism from the initial algebra to every other algebra behaviorally equivalent to it, so that the whole category is an equivalence class with respect to behavioral equivalence. If we restrict ourselves to static abstractions and to axiomatizations that define a reduced canonical model, then the set of all models satisfying the axioms is the same as the set of all models behaviorally equivalent to the canonical model, and our definition of correctness agrees with those used in the axiomatic approaches [37, 10, 9]. For the case where the canonical model defined by the axioms is not reduced, there is a lack of agreement on the proper definition of the set of implementations consistent with an algebraic specification [12, 9, 22].

6.2.2 Critical Comparison

The criteria for evaluating specification techniques given in [31] are: (1) formality, (2) constructibility, (3) comprehensibility, (4) minimality, (5) range of applicability, and (6) extensibility.

Both the algebraic technique and the abstract model technique as developed in this work are sufficiently formal, since both techniques have been given mathematical definitions.

Both techniques result in minimal specifications. It has often been (incorrectly) said that abstract model specifications are not minimal, because the model may have irrelevant characteristics. As our definition of correctness illustrates, only those properties of a model that are externally observable in terms of the operations of the abstraction are relevant, and those properties must be defined by any complete specification. Neither abstract model specifications nor algebraic specifications constrain either the representation structure or the algorithms that may be used by an implementation, as long as the externally observable behavior of the

abstraction is realized.

From a less formal point of view, it could be argued that the abstract representation is not directly observable in terms of the operations available to the user of the abstraction, and that this introduces the burden of keeping track of which details are directly observable and which details are not. The axiomatic approach has advantages with respect to this criterion, since there is no explicit mention of the representation. It has been shown [52] that there are abstractions that cannot be axiomatized without introducing auxiliary functions. Since the auxiliary functions also compute values that are not directly observable in terms of the operations, axiomatic specifications can also have details that need not appear in an implementation.

Another argument that has been used to suggest that abstract model specifications are not minimal is that the abstract representation tends to suggest an implementation. This is possible, but concern with issues of time and space efficiency often requires that the representation used in an implementation differ significantly from the representation used in the standard model, which is usually the simplest structure that will exhibit the desired behavior. The abstract representation is often defined in terms of mathematical structures not directly supported by the host programming language, so that in many cases it is not possible to use the specification structure in the implementation.

At the time of this writing, the abstract model technique has a clear advantage with respect to range of applicability over the algebraic specification technique, since it treats shared mutable data while the algebraic technique does not. We expect this advantage to be a temporary one, which will disappear as further research extends axiomatic specifications to apply to this domain also.

We have found abstract model specifications significantly easier to construct and to understand than algebraic specifications. This is a subjective impression based on our own experience, and we urge the reader to try both techniques and to form his or her own opinion. We conjecture that part of the reason for our experience is that the set of data objects is explicitly described by an abstract model specification, while it is implicitly defined by the interaction of a potentially large number of axioms in the algebraic technique. The result is that the operations can often be understood and defined one at a time and based on fairly local considerations when using the abstract model technique, whereas the interactions between a number of operations must be considered in the algebraic approach, requiring a more global analysis.

We have found that abstract model specifications are significantly easier to modify than algebraic specifications, especially in the case where the meaning of one operation is changed but the meaning of the abstract representation is not changed, because only the operation that is changed need be considered. In an algebraic specification, every axiom that mentions the operation that was changed must be reexamined, and usually each operation is mentioned in more than one axiom. The effort of extending the specification of an abstraction by adding a new operation is roughly the same as that required to define an operation in the initial design, and again we have found that the process is easier using the abstract model technique.

An algebraic specification can also be viewed as the definition of an abstract model whose representation is the word algebra, containing all of the expression that can be constructed from the names of the primitive operations. For abstractions whose operations are relatively easy to define using this representation (i.e., syntax trees), the algebraic specifications

are relatively simple, while for other abstractions the operations may be quite awkward to define using this representation, and an abstract model using a representation algebra with a significantly different structure may be much simpler than the corresponding algebraic specification. From this point of view, the abstract model technique is easier to use simply because it offers a wider choice of representation structures to start from. By using the fixpoint construction to define a representation domain of syntax trees, it is always possible to define an abstract model with essentially the same structure as any given algebraic definition.

Another criterion for judging a specification technique is the relative difficulty of checking whether a given specification is well formed. If we are interested in using specifications in the design process, it is helpful for the process of constructing the specifications to point out inconsistencies in the design, or at least to make them easier to find. We would like ill formed specifications to be easy to recognize.

To check that an abstract model specification is well formed it is necessary to check that the operations are well defined functions, and that the operations preserve the constraints adopted when defining the model. For each operation, it is necessary to check that the results of the operation satisfy the invariant relation specified by the **restrictions** section of the specification. It is also necessary to check that each operation will yield equivalent results when applied to either of two data objects related by the equivalence relation defined by the **identity** section of the specification. These properties are fairly easy to check informally, and they are generally not too difficult to prove rigorously. It is also usually fairly straightforward to check that the operations are defined for all inputs, and result in unique values. It is necessary to show that each invocation of an operation that can raise an exception will terminate in the expected termination condition, and that each recursive definition and each iota expression (see

Chapter 4) is well founded. Showing that a recursive function terminates is undecidable in the general case, but that seems to have little practical significance. In cases where something is wrong with the design, the designer will usually be unable to produce a function definition that even appears to be well formed.

In the algebraic approach, there is no analog to the data invariant, and the equivalence is guaranteed to be consistent with the operations by construction (of the canonical model). If an attempt is made to define an operation that attempts to produce different value for expressions representing equivalent abstract objects, then the result will be an inconsistent axiomatization, where the multiple values are redefined to be equivalent. In such a case the subordinate types of the canonical model often collapse into singleton sets. Incomplete definitions introduce extra data objects into the subordinate types, which are produced by expressions that cannot be reduced to bona fide elements of the subordinate types by the axiomatic definition. Rather than leading to an easily recognized failure, the algebraic technique will typically redefine the previously defined types in cases where the basic design is flawed.

Determining whether a given axiomatization is complete and consistent is generally acknowledged to be a difficult problem in modern mathematics, and there does not seem to be any straightforward procedure for checking the well formedness of an axiomatization. There are mechanical procedures for checking whether an axiomatization is complete and consistent that apply in restricted cases (the general problem is undecidable), but it is not clear whether these procedures can be used as practical aids in the design process.

An advantage of algebraic specifications is that fairly powerful automatic theorem provers for algebraically defined data abstractions have been developed. This advantage is

probably also temporary, pending the development of good domain specific theorem provers for the domains used to construct standard model specifications. For the domain of static data abstractions, it is possible to define an abstract model using equational axioms, by introducing an auxiliary operation that maps an object of the representation algebra into the abstract object it represents (cf. Hoare's abstraction function, [18]). Such an approach allows taking advantage of known properties of the modeling domain, and also of existing theorem provers for equational axiomatizations. It suffers from the disadvantage of not being immediately applicable to mutable data abstractions.

In our opinion abstract model specifications are clearly superior to algebraic specifications for the purpose of designing programs. The algebraic specification technique has advantages for the purpose of proving the correctness of programs at the current time, since it has been more extensively developed, but we feel that a long term advantage has not been demonstrated.

6.3 Directions for Future Research

One interesting question that has been raised but not resolved by the current work is whether or not abstract model specifications are better than axiomatic specifications with respect to program verification. Since the abstract representation of each data type must be considered when using abstract model specifications, and need not be considered when using axiomatic specifications, a naive analysis would indicate that proofs with respect to abstract model specifications are more complicated than the corresponding proofs with respect to axiomatic specifications, based on the sheer volume of detail to be expected. However, in the proofs we have done (manually), we have found that known properties of the modeling domain can often

be carried over to the abstract domain, leading to short and simple arguments. This phenomenon may have an analog for mechanical theorem provers, since special purpose theorem provers designed for the particular modeling domains used in constructing abstract models may be more efficient and more powerful than a general purpose theorem prover that must work with arbitrary axiomatizations.

If proofs of correctness are to be used for certifying software, then it is necessary to develop mechanical proof checking procedures, because proofs developed manually are at least as susceptible to errors as programs written by people. While a completely automated theorem proving facility would be nice to have, it looks likely that in a practical system the theorem prover will need human guidance, perhaps in the form of an informal outline of a proof, which the mechanical procedure tries to augment until it either discovers a formal proof or an error.

Our experience with proofs in terms of abstract model specifications indicates that an intuitive understanding of the model derived from familiarity with the underlying modeling domain often acts as a valuable guide to discovering a successful proof strategy. For axiomatic specifications this intuition is often lacking, and the process of trying to construct a proof degenerates into fairly blind symbol manipulation and syntax directed searching more often than for abstract model specifications. If the theorem prover must rely on human guidance, then the ease of finding intuitive insights can be an important consideration. We also conjecture that the extra structure provided by the abstract model is useful in constructing heuristics to guide the search strategy of a completely automated theorem prover.

In order to settle these questions, special purpose theorem provers oriented to the modeling domains used in abstract model specifications should be constructed and integrated into a program verification system.

Another question that is of interest is the extension of the framework developed here to incorporate nondeterminism and partial operations. Both of these extensions require a refinement of the idea of behavioral equivalence.

If the operations of a data abstraction can be nondeterministic, then a computation no longer has a unique value, but rather a set of possible values. Strict equivalence of the behavior of two models would require that the set of possible results of a computation be the same in both models. Since a more deterministic implementation of a nondeterministic operation is presumably correct if it always exhibits one of the possible behaviors for the standard model, an approximation relation that requires the set of possible results for the implementation to be a subset of the set of possible results for the standard model is a more appropriate metric for the correctness of an implementation, provided that the set of possible results is never empty.

Some abstractions have potentially useful operations that are inherently partial functions. One example is the domain of expressions for a Turing complete programming language, with an operation for evaluating expressions. In order to develop a model for such a structure, some sort of provision has to be made for cases in which the operations do not terminate. The impact of such an extension on the rest of the theory should be investigated.

Appendix I - Assumptions and Restrictions

1. Partial Operations

The operations that can be defined in a programming language are in general partial functions, since there may be circumstances under which they do not terminate. We will require the operations of a data abstraction to be total, because we feel that it is bad programming practice to design abstractions with primitive operations that may fail to terminate. Some recent work on specifying data abstractions with partial operations can be found in [27].

Many data abstractions have operations that make sense only for some proper subset of the input domain (ie. dividing by zero is not well defined). If an operation is invoked with arguments that are outside its natural domain of definition, the operation should terminate by raising an exception, to indicate that something unusual has happened. The reader should note that it is possible to transform a computable partial function into a computable total function that raises an exception only if the domain of definition of the partial function is a recursive set. An interpreter for any Turing-complete language has a domain of definition that is not recursive (otherwise the halting problem would be decidable), demonstrating that there are interesting functions that cannot be made to satisfy our restriction. Partial recursive procedures that compute such functions can of course be defined in terms of the primitive operations of a suitable data abstraction, but we do not allow them to be included as primitive operations of the abstraction.

2. Nondeterministic Operations

We would like to distinguish between partially specified operations and nondeterministic operations. A partially specified operation is defined only for some proper subset of its input type, and presumably the designer does not care what the operation does if it is presented with an input outside of that subset. We feel that it is bad design practise to produce specifications of this type, because of the possibility of undetected errors in the use of the abstraction. The only case in which we truly do not care what an operation does on a certain input is if we know that it will never be called with that input. A well designed data abstraction should raise an exception for all inputs for which no normal response is specified, so that attempts to use the operation outside of its domain of validity will not pass undetected.

Data abstractions with nondeterministic operations are potentially interesting, but are not treated in the main body of this work. An operation can be described by an input-output relation R, which relates the inputs of an operation to the legal output values for those inputs. For a deterministic operation, such a relation is single valued, and is in fact an ordinary function. Some operations are most naturally described by relations that are not single valued: the programmer wants the operation to satisfy certain criteria (eg. the relation R), and does not care if there is a unique result, or which valid result is actually chosen, if there is more than one valid choice. We do not recommend introducing extra constraints with the sole purpose of restricting R to the point where it becomes a function. Such constraints complicate the specification by introducing irrelevant details, and also may exclude some of the simplest and most efficient implementations, which would be perfectly acceptable without the artificial constraints.

A non functional input-output relation R is consistent with a whole class of operations, some of which are deterministic, and some of which are not. The reader should note that it is quite possible to implement a data abstraction with nondeterministic operations on a deterministic machine, because an abstract data object need not have a unique representation in the implementation. For example, consider the data abstraction consisting of the finite sets of natural numbers, together with the usual set theoretic operations, and a *choose* operation. The *choose* operation returns an element of a given set if the set is nonempty, and raises an exception otherwise. It is not specified which element of the set is to be chosen if there is more than one. Abstract sets are immutable, and two sets are equal if and only if they have the same elements. In an implementation, sets might be represented as linked lists, and the *choose* operation might return the first element in the list. However, since there are many different representations for the same set, with the elements stored in different orders, the *choose* operation appears to be nondeterministic when viewed as an operation on abstract sets.

We know of no work that has been done on specifying data abstractions with nondeterministic operations. Some work on specifying nondeterministic operations in terms of relations is reported in [34].

3. Concurrency

Concurrent access to data objects by parallel processes is an interesting subject that is beyond the scope of this Thesis. It is profitable to consider parallel processing in the context of data abstractions [20, 16, 6, 38, 24] because processes need to be synchronized only if they operate on shared data. Even though a quite a bit of work has been done in this area, the issues involved in specifying the correctness of a data abstraction in the presence of concurrent

mutation of data objects are not yet well understood.

4. Exceptions

Since there is no generally accepted model of exceptions and exception handling, we have chosen a point of view that simplifies the interface presented by an operation, and which helps to separate the externally visible behavior of an operation from the internal processes that produce that behavior.

We assume that an operation terminates whenever it raises an exception. Thus an operation may terminate in any one of a number of conditions, one of which is normal and the rest of which are exceptional. In general, the results of the operation in each condition will be different, and must be specified for all possible termination conditions in a complete description of the operation.

The alternative to our point of view is to allow an exception to cause some events, and then to continue performing the original operation at the point where it left off. This alternative is not attractive because the separation between the specifications of an operation and the details of its implementation breaks down. Given a specification of an operation that describes the results of the operation for the normal termination condition and gives the conditions under which each exception occurs, and given a specification of an exception handler for each exception raised by the operation, we still do not have enough information to predict the behavior of the operation in the context of the specified exception handlers. It is necessary to analyze the implementation of the operation with respect to the specifications of the exception handlers in order to determine the effects of the operation. Since different invocations of the operation can occur in the contexts of different exception handlers, we cannot

treat an operation as a closed module if we adopt the resumption model of exception handling. Exceptions are discussed further in Chapter 2.

5. Own Data

We assume that the operations of a data abstraction are functional. This means that an operation must not have any internal state, so that the results of an operation depend only on the information contained in the data objects passed to the operation as arguments (which may include references to other objects). Data objects may themselves have states, so that we are not excluding the possibility that an operation may return different results if it is invoked with the same arguments at two different times. This restriction is meant to prohibit type managers (ie. SIMULA classes, CLU clusters, ALPHARD forms, etc.) from keeping mutable own data, which introduces a component of the state associated with the type as a whole, rather than with the individual data objects. This issue is discussed further in Section 3.2.

Appendix II - Basic Type Definitions

The definitions of the natural numbers and the integers are imported directly from the underlying standard mathematics. The definition of the natural number abstraction is shown in Figure 24. As in the definition of **set** in Chapter 4, the standard notations for natural numbers and integers are used in the definitions of the operations to refer to the standard operations of the underlying mathematical domains, while the same notations are introduced as abbreviations for the operations of the exception algebra, for use in the definitions of other modules. The only nonstandard feature of this definition of the natural

---

Figure 24. Natural Numbers

type nat as NN

with	constant[n]:	$\rightarrow$ NN	**as** n	for $n \in$ **N**
	zero:	$\rightarrow$ NN	**as** 0	
	successor:	NN $\rightarrow$ NN	**as** σ(**arg** 1)	
	plus:	NN x NN $\rightarrow$ NN	**as arg** 1 + **arg** 2	
	times:	NN x NN $\rightarrow$ NN	**as arg** 1 $\diamond$ **arg** 2	
	less:	NN x NN $\rightarrow$ **boolean**	**as arg** 1 < **arg** 2	
	equal:	NN x NN $\rightarrow$ **boolean**	**as arg** 1 = **arg** 2	

representation	natural numbers **N**
restrictions	none
identity	natfequal

operations	constant[n]() = n
	zero() = 0
	successor(x) = σ(x)
	plus(x, y) = x + y
	times(x, y) = x $\diamond$ y
	less(x, y) = x < y

end nat

numbers is the infinite parameterized family of constants. These operations are introduced so that we can use the familiar decimal notation for natural numbers in our specification language, rather than having to build up each number from zero using the successor function, which quickly gets cumbersome.

The definition of the integers is shown in Figure 25. Integers also have an infinite supply of constant operations. Note the conversion operations *integer* and *nn*, which serve to convert integers to natural numbers and vice versa. The *quotient* and *remainder* operations have exception conditions in the cases where the standard mathematical definitions are undefined. The *quotient* operation rounds down irrespective of the sign of its arguments, in agreement with the usual mathematical definition, and in contrast to the way division works in most programming languages (e.g., FORTRAN).

The astute reader will have noticed that we have omitted the definitions of the operations $>$, $\neq$, $\leq$, and $\geq$, even though we have used them freely in the specification language. The astute reader will also be able to supply the standard definitions for these operations, and is advised to do so.

These types are intended for use in the specification language. The corresponding types for a programming language should probably be designed differently, to include limitations on the sizes of the numbers, exception conditions for cases in which those size limitations are exceeded, and additional operations for converting strings of decimal digits into numbers, and for printing out numbers.

Figure 25. Integers

type int **as** I

with	constant[*n*]:	→ I	**as** *n*	for *n* ∈ Z
	integer:	nat → I		
	minus:	I → I	**as** - **arg** I	
	plus:	I × I → I	**as arg** I + **arg** 2	
	difference:	I × I → I	**as arg** I + **arg** 2	
	times:	I × I → I	**as arg** I * **arg** 2	
	quotient	I × I → I + ⟨zero_divide : ⟩		
	remainder:	I × I → I + ⟨zero_divide : ⟩		
	abs:	I → I	**as** | **arg** I |	
	nn:	I → nat + ⟨wrong_sign : ⟩		
	less:	I × I → boolean	**as arg** I < **arg** 2	
	equal:	I × I → boolean	**as arg** I = **arg** 2	

representation integers Z
restrictions none
identity int$equal

operations
constant[*n*]() = *n*
integer(n) = n in Z
minus(x) = -x
difference(x, y) = x - y
times(x, y) = x * y
quotient(x, y) = if y = 0 then ⟨zero_divide : ⟩
else q : ∃r[x = q * y + r & 0 ≤ r < abs(y)]
remainder(x, y) = if y = 0 then ⟨zero_divide : ⟩
else r : ∃q[x = q * y + r & 0 ≤ r < abs(y)]
abs(x) = if x < 0 then -x else x
nn(x) = if x < 0 then ⟨wrong_sign : ⟩ else x in N
less(x, y) = x < y
equal(x, y) = x = y

end int

Appendix III - Proofs

An exception algebra differs from a heterogeneous algebra as defined in [1] by having a disjoint union structure for the ranges of the operations, where the disjoint union is indexed by termination conditions, and where the components of the disjoint union are cartesian products of the phyla. In a heterogeneous algebra, the range of each operation has to be some phylum of the algebra. The definitions of basic algebraic concepts such as subalgebras, congruence relations, quotient structures, and homomorphisms have to be adapted slightly to fit into our framework. The required extensions are concerned mostly with termination conditions. For example, a congruence relation is an equivalence relation that preserves all of the operations of an exception algebra, so that if corresponding arguments of an operation are related by the congruence, then the termination conditions of the two invocations must be identical, and corresponding return values must be related by the components of the congruence relation for the appropriate phyla. As in [1], an equivalence relation on an exception algebra is defined to be an indexed set of equivalence relations, one for each phylum.

Theorem 2: Every equivalence class of static models with respect to the behavioral equivalence relation contains a reduced model.

Proof: Let E be an equivalence class of models with respect to behavioral equivalence,
and choose $M \in E$. This will always be possible,
since equivalence classes are nonempty by definition.
Let M' be the subalgebra of M containing only the reachable objects of the principal type,
and with the same subordinate types as M.
M' is closed with respect to the operations of M, since it contains all reachable data objects.
M' is behaviorally equivalent to M
since the value of any closed computation C in M' is the same as the value of C in M.
Let $M'' = M'/\equiv$, where $\equiv$ is the external equivalence relation defined in Chapter 3.

M'' is well defined because ≡ is consistent with all of the operations by construction.
Then M'' is reduced and behaviorally equivalent to M.
Every element of the principal type of M'' is reachable,
because any such element is equal to $[x]$ for some x in the principal type of M',
and every such x is reachable, by the construction of M'.
Any two elements of the principal type of M'' that are externally equivalent must be identical,
by the construction of M'' from M'.
Hence M'' is reduced.
M'' is behaviorally equivalent to M' because it is a homomorphic image of M',
under the natural homomorphism h defined by $h(x) = [x]$ if $x \in d$ and $h(x) = x$ otherwise,
where d is the principal type of M'.
Since behavioral equivalence is transitive,
M'' is behaviorally equivalent to M,
and the theorem is established.
End of Proof

Theorem 3: If two reduced models are behaviorally equivalent, then they are isomorphic.

Proof: Let $M1$ and $M2$ be reduced and behaviorally equivalent.
Define the isomorphism f as follows.
For every closed computation C, let $f(\text{value}(C, M1)) = \text{value}(C, M2)$.
By Lemma 1 below, whenever $\text{value}(C, M1) = \text{value}(C', M1)$
then $\text{value}(C, M2) = \text{value}(C', M2)$,
so that f is single valued, and hence a function.
The inverse mapping is obtained by interchanging $M1$ and $M2$ in the above definition,
and it is also single valued, by the same argument.
So f is 1 : 1.
The operations of the algebra are preserved by construction,
so the isomorphism is established.
End of Proof

Lemma 1: Let $M1$ and $M2$ be behaviorally equivalent exception algebra models, let C and C' be closed computations, and let $\text{value}(C, M1) = \text{value}(C', M1)$. Then $\text{value}(C, M2)$ is externally equivalent to $\text{value}(C', M2)$.

Proof: let $M1$ and $M2$ be behaviorally equivalent exception algebras,
let C and C' be closed computations,
let value(C, $M1$) = value(C', $M1$),
and let $C0$ be an open computation.
Then value($C0$, value(C'', M), M) = value(C''‖$C0$, M),
for any exception algebra M,
where C''‖$C0$ is the concatenation of the computations C'' and $C0$[2 .. length($C0$)],
and where the step indices of all of the argument specifications in $C0$
have been increased by length(C'')-1.
then value($C0$, value(C, $M2$), $M2$) =
value(C‖$C0$, $M2$) = by the definition of concatenation,
value(C‖$C0$, $M1$) = since $M1$ and $M2$ are behaviorally equivalent
value($C0$, value(C, $M1$), $M1$) = by the definition of concatenation
value($C0$, value(C', $M1$), $M1$) = by assumption,
value(C'‖$C0$, $M1$) = by the definition of concatenation,
value(C'‖$C0$, $M2$) = since $M1$ and $M2$ are behaviorally equivalent,
value($C0$, value(C', $M2$), $M2$) by the definition of concatenation.
So value(C, $M2$) is externally equivalent to value(C', $M2$).
End of Proof

Theorem 4: If M is behaviorally equivalent to M' and M is reduced, then there is a homomorphism from a subalgebra of M' onto M.

Proof: Let M'' be the subalgebra of M' containing only the reachable
objects of the principal type of M',
and with the same subordinate types as M'.
The quotient of M'' with respect to the external equivalence relation is reduced,
and behaviorally equivalent to M' by **Theorem** 2,
and by transitivity of behavioral equivalence, it is also behaviorally equivalent to M.
Then by **Theorem** 3, the quotient is isomorphic to M.
The composition of the natural homomorphism from M'' to the quotient and
the isomorphism guaranteed by **Theorem** 3 is a homomorphism from M'' to M,
so the theorem is established.
End of Proof

Theorem 5: Every chain of algebras with respect to $\sqsubseteq$ has a least upper bound.

Proof: Let $A_i : i \in \mathbf{N}$ be a chain of algebras with respect to $\sqsubseteq$.
Then $A = \bigcup_{i \in \mathbf{N}} A_i$, where A is defined as follows.

$\forall \alpha \in A.\text{typenames}\ [\ A.\text{phyla}_\alpha = \bigcup_{i \in \mathbf{N}} A_i.\text{phyla}_\alpha\ ]$

$\forall \beta \in A.\text{opnames}\ [\ A.\text{operations}_\beta = \bigcup_{i \in \mathbf{N}} A_i.\text{operations}_\beta\ ]$

$A.x = \bigcup_{i \in \mathbf{N}} A_i.x$,
where x can be any one of the following components:
typenames, opnames, tcnames, arglength, argtype, tc, rlength, rtype, or pt.

By Lemma 2 the operations and type description functions are well defined.

$A_i \sqsubseteq A$ for all $i \in \mathbf{N}$.
since $s_j \subseteq \bigcup_{i \in \mathbf{N}} s_j$ for any $j \in \mathbf{N}$.
So A is an upper bound for the chain A_i.
If $A_i \sqsubseteq B$ for all $i \in \mathbf{N}$ then $A \sqsubseteq B$,
since $s_i \subseteq S$ for all $i \in \mathbf{N}$ implies $\bigcup_{i \in \mathbf{N}} s_i \subseteq S$.
So A is the least upper bound for the chain A_i.
End of Proof

Lemma 2: If $f_i : i \in \mathbf{N}$ is a chain of functions with respect to $\sqsubseteq$, then $f = \bigcup_{i \in \mathbf{N}} f_i$ is a well defined function.

Proof: We have to show that $f = \bigcup_{i \in \mathbf{N}} f_i$ is single valued.
Proof by contradiction.
Suppose f is not single valued.
Then for some x, $\langle x, a\rangle \in f$ and $\langle x, b\rangle \in f$ where $a \neq b$.
Since $f = \bigcup_{i \in \mathbf{N}} f_i$,
pick n, m such that $\langle x, a\rangle \in f_n$ and $\langle x, b\rangle \in f_m$.
Since f_i is a chain, $f_n \subseteq f_{\max(n, m)}$ and $f_m \subseteq f_{\max(n, m)}$.
So $\langle x, a\rangle \in f_{\max(n, m)}$ and $\langle x, b\rangle \in f_{\max(n, m)}$ where $a \neq b$.
But f_i is a single valued function for all $i \in \mathbf{N}$, contradiction.

So f must be single valued.
End of Proof

Note that we are treating a function f as the set of all pairs $\langle x, f(x) \rangle$ such that $x \in \text{domain}(f)$.

Theorem 6: The tuple transformation is continuous with respect to $\sqsubseteq$.

Proof: Let A_i be a chain with respect to $\sqsubseteq$.
Let $\sqcup$ denote the least upper bound with respect to $\sqsubseteq$.
$(\bigcup_{i \in \mathbf{N}} A_i) \times S_2 \times \ldots \times S_n = \bigcup_{i \in \mathbf{N}} (A_i \times S_2 \times \ldots \times S_n)$
from the definition of union and cross product.

The definition of each operation is a functional F from the phyla to operations on the phyla, with the property that the value of an operation on any input depends only on the input values, and not on the phylum as a whole.
(The finite quantification in the definition of equal can be expanded into an equivalent finite conjunction.)
So $F(\bigcup S_i)(x) = F(S_j)(x)$ for any S_j such that $x \in S_j$
$F(S_j)(x)$ is undefined if $\neg\, x \in S_j$
So $F(\bigcup S_i)(x) = \bigcup F(S_j)(x)$.
The definitions of the signature functions also have this property.
So the tuple transformation on algebras is continuous with respect to $\sqsubseteq$.
End of Proof

Theorem 7: Let $M1$ and $M2$ be complete exception-algebra models with the same signature and the same interpretations for the subordinate types, and let h be a homomorphism from $M1$ to $M2$, such that h is the identity mapping on all of the subordinate types. Then $M1$ and $M2$ are behaviorally equivalent.

Proof: For every finite closed computation C, we have to show that:

A. C is feasible in $M1$ if and only if C is feasible in $M2$.

B. value(C, $M1$) = value(C, $M2$) whenever C is feasible in $M1$ and produces a boolean value.

Let $H(C)$ = (feasible(C, $M1$) ≡ feasible(C, $M2$)) &
(length(C) ≥ 1 & feasible(C, $M1$)) ⇒ h(value(C, $M1$)) = value(C, $M2$).
Assuming that $H(C')$ holds for all C' such that length(C') < length(C),
show that $H(C)$ holds.

Case 1: length(C) = 0

$H(C)$ is trivially true, since the antecedent of the implication is false.
A. holds since the empty computation is feasible in any model.
B. holds since there are no computations of length 0 producing a boolean value.

Case 2: length(C) > 0

Let length(C) = n and let C' = C[1 .. n-1].
Then $H(C')$ since length(C') = n-1 < length(C).

A. To show feasible(C, $M1$) if and only if feasible(C, $M2$)

Case 2.1: C' is not feasible in $M1$.

Then by the induction hypothesis $H(C')$, C' is not feasible in $M2$.
Since C' is a prefix of C, C is not feasible in $M1$ or $M2$.
So A. holds for case 2.1.

Case 2.2: C' is feasible in $M1$.

Then by the induction hypothesis C' is feasible in $M2$.
Therefore the termination conditions of the arguments match the requirements for every step of C' in both models.
C is feasible in $M1$ if and only if the termination conditions of the arguments of $C[n]$ match the requirements of step $C[n]$.
Each argument x_i is the value of C_i,
where length(C_i) ≥ 1 and where C_i is a proper prefix of C.
By the induction hypothesis h(value(C_i, $M1$)) = value(C_i, $M2$).
Then tc(h(value(C_i, $M1$)) = tc(value(C_i, $M2$)),
since homomorphisms preserve termination conditions.
Therefore the arguments will match the requirements for the interpretation of C in $M2$ whenever they will match for the interpretation of C in $M2$.
So A. is established for case 2.2.

B. Assume C is feasible in $M1$ and length(C) ≥ 1.
Show h(value(C, $M1$)) = value(C, $M2$).

Each argument x_i of the last operation of C is the result of some prefix C_i of C,
where 1 ≤ length(C_i) < length(C).
By the induction hypothesis, h(value(C_i, $M1$) = value(C_i, $M2$).
Since h is a homomorphism, h preserves the operations of $M1$ and $M2$.
So h(value(C, $M1$)) = value(C, $M2$).

So $H(C)$ for all computations C.
If the principal type of $M1$ is boolean then $M1$ = $M2$,

since there is a unique standard model for the boolean domain,
and otherwise boolean is a subordinate type.
In either case, $h_{boolean}$ is the identity mapping.
So if a computation results in a boolean value,
it must result in the same boolean value in $M1$ and in $M2$.
So $M1$ and $M2$ are behaviorally equivalent.

Then $H(C)$ holds for all finite computations C.
End of Proof

Theorem 8: Let $M1$ be a state machine model and let $M2$ be an exception algebra model with the same signature and the same interpretations for the subordinate types. Let c be a correspondence function from $M1$ to $M2$, such that c returns its second argument for all subordinate types. Then $M1$ and $M2$ are behaviorally equivalent.

Proof: For every finite closed computation C, we have to show that:

A. C is feasible in $M1$ if and only if C is feasible in $M2$.

B. value(C, $M1$) = value(C, $M2$) whenever C is feasible in $M1$ and produces a boolean value.

Let state(C, M) denote the final state produced by
the interpretation of the closed computation C in the state machine model M.

Let $H(C)$ = (feasible(C, $M1$) ≡ feasible(C, $M2$)) &
(length(C) ≥ 1 & feasible(C, $M1$)) ⟹ c(state(C, $M1$), value(C, $M1$)) = value(C, $M2$).
Assuming that $H(C')$ holds for all C' such that length(C') < length(C),
show that $H(C)$ holds.

Case 1: length(C) = 0

> $H(C)$ is trivially true, since the antecedent of the implication is false.
> A. holds since the empty computation is feasible in any model.
> B. holds since there are no computations of length 0 producing a boolean value.

Case 2: length(C) > 0

> Let length(C) = n and let C' = C[1 .. n-1].
> Then $H(C')$ since length(C') = n-1 < length(C).

A. To show feasible(C, $M1$) if and only if feasible(C, $M2$)

Case 2.1: C' is not feasible in $M1$.

Then by the induction hypothesis $H(C')$, C' is not feasible in $M2$.
Since C' is a prefix of C, C is not feasible in $M1$ or $M2$.
So A. holds for case 2.1.

Case 2.2: C' is feasible in $M1$.

Then by the induction hypothesis C' is feasible in $M2$.
Therefore the termination conditions of the arguments match the requirements
for every step of C' in both models.
C is feasible in $M1$ if and only if the termination conditions of the arguments of $C[n]$
match the requirements of step $C[n]$.
Each argument x_i is the value of C_i,
where length(C_i) ≥ 1 and where C_i is a prefix of C.
By the induction hypothesis c(state(C_i, $M1$), value(C_i, $M1$)) = value(C_i, $M2$).
Then tc(c(state(C_i, $M1$), value(C_i, $M1$)) = tc(value(C_i, $M2$)),
since correspondence functions preserve termination conditions.
Therefore the arguments will match the requirements for
the interpretation of C in $M2$
whenever they will match for the interpretation of C in $M2$.
So A. is established for case 2.2.

B. Assume C is feasible in $M1$ and length(C) ≥ 1.
Show c(state(C, $M1$), value(C, $M1$)) = value(C, $M2$).

Each argument x_i of the last operation of C is the result of some prefix C_i of C,
where 1 ≤ length(C_i) < length(C).
By the induction hypothesis, c(state(C_i, $M1$), value(C_i, $M1$)) = value(C_i, $M2$).
Since C_i is a prefix of C, c(state(C_i, $M1$), x_i) = c(state(C, $M1$), x_i),
by the monotonicity property of correspondence functions.
Since c is a correspondence function, c preserves the operations of $M1$ and $M2$.
So c(state(C, $M1$), value(C, $M1$)) = value(C, $M2$).

So $H(C)$ for all computations C.
By the hypothesis of the theorem, c is the identity mapping on the boolean domain.
So if a computation results in a boolean value,
it must result in the same boolean value in $M1$ and in $M2$.
So $M1$ and M are behaviorally equivalent.

Then $H(C)$ holds for all finite computations C.
End of Proof

Appendix IV - Syntax

The syntax of an abstract model specification is given below in an extended form of bnf. [X] means that X is optional. Large parentheses ( ) are symbols of the meta language used for grouping terms. Small parentheses and square brackets "(", ")", "[", and "]" are terminal symbols denoting the respective characters themselves. X^* means X can be repeated zero or more times. X^+ is the same as X X^* (X may occur one or more times).

<specification>::= <module> | <type definition>
<module>::= **module** <type definition>$^+$ **end module**

<type definition>::= **type** <type name> [<parameter list>] [<abbreviation>]
[<requires>]
<signature>
<rep spec>
<ops>
[<auxiliary signature>]
[<definitions>]
end <type name>

<parameter list>::= [<parameter name> (, <parameter name>)*]
<abbreviation>::= **as** <abbreviation body>

<requires>::= **requires** <parameter type> (, <parameter type>)*
<parameter type>::= <parameter name> : <type name> [**such that** <predicate>]

<signature>::= **with** <function type>$^+$
<auxiliary signature>::= **internal** <function type>$^+$
<function type>::= <function name> : [<domain spec>] → [<domain spec>] <condition spec>*
<domain spec>::= <type name> (× <type name>)*
<condition spec>::= + ⟨ <exception name> : [<domain spec>] ⟩

<rep spec>::= <domain equation>.[<restriction>][<equivalence>]
<domain equation>::= <domain name> = <domain expression>
<domain expression>::= <domain name> | { <domain name>* }
| tuple[[<labeled expression list>]]
| union[[<labeled expression list>]]
| set[<domain expression>]
| sequence[<domain expression>]
<labeled expression list>::= <labeled expression> (, <labeled expression>)*
<labeled expression>::= <label> : <domain expression>
<restriction>::= **restrictions** none | **restrictions** <identifier> such that <predicate>
<equivalence>::= **identity** <operation name>

<ops>::= **operations** <operation definition>+
<definitions>::= **definition** <operation definition>+
<operation definition>::= <operation name> <argument list> = <operation body> [<locals>]
<argument list>::= [(<identifier>)] (<identifier>*)
<operation body>::= <identifier> | <operation name> <expression list>
| <identifier> : <predicate>
| if <boolean expression>
then <operation body>
else <operation body>
<expression list>::= () | (<operation body> (, <operation body>)*)
<locals>::= where (<variable> = <operation body>)+

The grammar shown above specifies only the context free part of the language. There are a number of additional constraints that must be met for a well formed specification. For example, the number of argument expressions to an operation in an operation body must be the same as the number of type names in the domain specification of the operation in the signature.

References

1. Birkhoff, G. and Lipson, J. D. "Heterogeneous Algebras" *Journal of Combinatorial Theory 8*, 115-133, (1970).

2. Burstall, R. "Some Techniques for Proving Correctness of Programs which Alter Data Structures", *Machine Intelligence 7*, 23-50, Halstead Press (1972).

3. Curry, H. B. and Feys, R. *Combinatory Logic*, North-Holland, 1958.

4. Dahl, O-J. and Hoare, C. A. R. "Hierachical Program Structures", *Structured Programming*, A. P. I. C. Studies in Data Processing, No. 8, Academic Press, 1972, 175-220.

5. Dahl, O-J., Myhrhaug, B., and Nygaard, K. "The Simula 67 Common Base Language", Publication No. S-22, Norwegian Computing Center, Oslo, 1970.

6. Flon, L. and Habermann, A. N. "Towards the Construction of Verifyable Software Systems", *Proc. Conf. on Data: Abstraction, Definition, and Structure*; also in *ACM SIGPLAN Notices 8*, 2 (1976).

7. Goguen, J. A., Thatcher, J. W., Wagner, E. G. and Wright, J. B. "Abstract Data Types as Initial Algebras and the Correctness of Data Representations", *Proc. of the Conference on Computer Graphics, Pattern Recognition, and Data Structures*, 89-93, 1975.

8. Goguen, J. A. "Abstract Errors for Abstract Data Types", *Formal Description of Programming Concepts*, E. Neuhold, ed., North-Holland, 1978, 491-522. (Proceedings of the IFIP Working Conference on Formal Description of Programming Concepts, Aug. 1977.)

9. Goguen, J. A., Thatcher, J. W., Wright, E. G. "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types", *Current Trends in Programming Methodology*, Vol. 4, Data Structuring, R. T. Yeh, ed., Prentice Hall, Englewood Cliffs, New Jersey, 1978.

10. Guttag, J. V. *The Specification and Application to Programming of Abstract Data Types*, Ph. D. Thesis, University of Toronto CSRG-59 (1975).

11. Guttag, J. V., Horowitz, E., and Musser, D. R. "Abstract Data Types and Software Validation", *Comm. of the ACM, Vol. 21* No. 12 (Dec. 1978), 1048-1064.

12. Guttag, J. V. and Horning, J. J. "The Algebraic Specification of Abstract Data Types", *Acta Informatica 10*, No. 1, 27-52 (1978).

13. Guttag, J. V. "Notes on Type Abstraction", *Proc. IEEE Conf. on Specifications of Reliable Software*, 36-46, April 1979.

14. Harel, D. *Logics of Programs: Axiomatics and Descriptive Power*, MIT Ph. D. Thesis, May 1978.

15. Hewitt, C. and Baker, H. "Actors and Continuous Functionals", *Formal Description of Programming Concepts*, E. Neuhold, ed., North-Holland, 1978, 367-387. (Proc. IFIP Working Conference on Formal Description of Programming Concepts, Aug. 1977.)

16. Hewitt, C. and Atkinson, R. R. "Parallelism and Synchronization in Actor Systems", Record of the 1977 Conference on Principles of Programming Languages, January 1977, 267 - 280.

17. Hoare, C. A. R. Procedure and Parameters - An Axiomatic Approach", *Symposium on the Semantics of Algorithmic Languages", E. Engler,, Ed., Springer Verlag, 1971.*

18. Hoare, C. A. R. "Proof of Correctness of Data Representations", *Acta Informatica 1*, 4 (1972), 271-281.

19. Hoare, C. A. R. "Notes on Data Structuring", *Structured Programming*, A. P. I. C. Studies in Data Processing, No. 8, Academic Press, 1972, 83-174.

20. Hoare, C. A. R. "Monitors: An Operating System Structuring Concept", *Comm. of the ACM 17*, 10 (Oct. 1974).

21. Jones, A. K. and Liskov, B. H., "A Language Extension for Controlling Access to Shared Data", *IEEE Transaction on Software Engineering, Vol. SE-2*, No. 4, (Dec. 1976), 277-285.

22. Kapur, D. "Towards a Theory for Data Abstractions", Ph. D. thesis proposal, MIT, Feb. 1979.

23. Kleene, S. C. *Introduction to Metamathematics*, Van Nostrand, 1950.

24. Laventhal, M. S. *Synthesis of Synchronization Code for Data Abstractions*, forthcoming MIT Ph. D. Thesis.

25. Lehmann, D. J., *Categories for Fixpoint Semantics*, Ph. D. Thesis, University of Warwick, Theory of Computation Report 15, 1976.

26. Lehmann, D. J. "Modes in Algol Y", Proceedings 5th I. I. I. Conference on Implementation and Design of Algorithmic Languages, Guidel, France, May 1977.

111-123.

27. Lehmann, D. J. and Smyth, M. B. "Data Types", *Proc. 18-th IEEE Symposium on Foundations of Computer Science*, Nov. 1977, 7-12.

28. Liskov, B. H. and Berzins, V., "An Appraisal of Program Specifications", in *Research Directions in Software Technology*, MIT Press, Cambridge, Mass., 1979.

29. Liskov, B. H., Snyder, A., Atkinson, R. R., Schaffert, J. C. "Abstraction Mechanisms in CLU", *Comm. of the ACM 20*, 8 (Aug. 1977), 564-576.

30. Liskov, B. H. and Snyder, A. "Structured Exception Handling", 1977, MIT Computation Structures Group Memo 155.

31. Liskov, B. H. and Zilles, S. "Specification Techniques for Data Abstractions", *IEEE Transactions on Software Engineering, Vol. SE-1*, 7-19 (1975).

32. Luckham, D. C. and Suzuki, N. "Automatic Program Verification V: Verification Oriented Proof Rules for Arrays, Records, and Pointers", Stanford AIM-278, (March 1976).

33. McCarthy, J. "A Basis for a Mathematical Theory of Computation", *Computer Programming and Formal Systems*, Braffort and Hirschberg, eds., North Holland Publishing Co., Amsterdam-london, 1963.

34. Milner, R. "An Algebraic Definition of Simulation between Programs", *Proc. Joint Conf. on Artificial Intelligence*, 481-489, 1971.

35. Musser, D. R. "A Data Type Verification System Based on Rewrite Rules", *Proc. of the Sixth Texas Conf. on Computing Systems*, Austin, Texas, Nov. 1977.

36. Musser, D. R. "Abstract Data Type Specification in the AFFIRM System", *Proc. of the Specifications of Reliable Software Conf.*, IEEE Computer Society, Technical Committee on Software Engineering, April 1979, 47-57.

37. Nakajima, R., Honda, M. and Nakahara, H. "Describing and Verifying Programs with Abstract Data Types", *Formal Description of Programming Concepts*, E. Neuhold, ed., North-Holland, 1978, 527-556. (Proc. IFIP Working Conference on Formal Description of Programming Concepts, Aug. 1977.)

38. Owicki, S. "Verifying Concurrent Programs with Shared Data Classes", *Formal Description of Programming Concepts*, E. Neuhold, ed., North-Holland, 1978, 279-298. (Proc. IFIP Working Conference on Formal Description of Programming Concepts, Aug. 1977.)

39. Palme, J. "Protected Program Modules in SIMULA 67", FOAP C8372-M3(E5), Operations Research Center, Research Institute of National Defense, Stockholm, Sweden (1973).

40. Parnas, D. L. "On the Criteria To Be Used in Decomposing Systems into Modules", *CACM 15*, 12, 1053-1058 (December 1972).

41. Polajnar, J. *An Algebraic View of Protection and Extendability in Abstract Data Types*, Ph. D. Thesis, University of Southern California, September 1978.

42. Pratt, V. "Semantical Considerations on Floyd-Hoare Logic", MIT/LCS/TR-168, also in *Proc. IEEE Symposium on Foundations of Computer Science*, Oct. 1976.

43. Schaffert, J. C. *A Formal Definition of CLU*, MIT Master's Thesis, January 1978, also MIT/LCS/TR-193.

44. Schaffert, J. C. *Specifying Meaning in Object Oriented Languages*, MIT Ph. D. Thesis (forthcoming).

45. Scheifler, R. W. *A Denotational Semantics of CLU*, MIT Master's Thesis, also MIT/LCS/TR-201, May 1978.

46. Scott, D., "Data Types as Lattices", *SIAM Journal on Computing, Vol. 5*, No. 3, (Sept. 1976), 522-587.

47. Shaw, M. "Abstraction and Verification in ALPHARD: Design and Verification of a Tree Handler", Computer Science Dept., Carnegie-Mellon University, June, 1976.

48. Shoenfield, J. *Mathematical Logic*, Addison-Wesley, Reading, Massachusetts, 1967.

49. Stoy, J. *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, Mass., 1977.

50. Strachey, C. "The Varieties of Programming Language", Technical Monograph PRG-10, Programming Research Group, Oxford University Computing Laboratory, March 1973.

51. Suzuki, N. *Automatic Verification of Programs with Complex Data Structures*, Ph. D. thesis, Stanford University, 1976.

52. Thatcher, J. "Data Type Specification: Parameterization and the Power of Specification Techniques", *Proceedings, SIGACT 10-th Annual Symposium on Theory*

of Computing, San Diego, California, (May 1978), 119-132.

53. Wegbreit, B. and Spitzen, J. M. "Proving Properties of Complex Data Structures", *Journal of the ACM Vol. 23*, No. 2 (April 1976), 389-396.

54. Wulf, W., London, R., and Shaw, M. "Abstraction and Verification in ALPHARD: Introduction to Language and Methodology", 1976, Carnegie-Mellon University Technical Report; also USC ISI Research Report.

55. Yonezawa, A. *Specification and Verification Techniques for Parallel Programs based on Message Passing Semantics*, 1977, MIT Ph. D. Thesis.

56. Zilles, S. "Algebraic Specification of Data Types", *Project MAC Progress Report* 1974, 52-58; also MIT Computation Structures Group Memo 119.

This empty page was substituted for a blank page in the original document.

CS-TR Scanning Project
Document Control Form

Date : 10 / 19 / 95

Report # LCS-TR-221

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 176 (183-images)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☐ Single-sided or
☒ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☒ Unknown ☐ Other:_____

Check each if included with document:

☐ DOD Form ☐ Funding Agent Form ☒ Cover Page
☒ Spine ☒ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages(by page number):_____

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description :	Page Number:
IMAGE MAP: (1-176) UNN# TITLE PAGE, 2-175, UNN# BLANK	
(177-183) SCANCONTROL, COVER, SPINE, PRINTER'S NOTES, TRGT'S (3)	

Scanning Agent Signoff:

Date Received: 10 / 19 / 95 Date Scanned: 10 / 30 / 95 Date Returned: 11 / 2 / 95

Scanning Agent Signature: Michael N. Cook

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**